

25

Programowanie systemów wbudowanych

„»Niebezpieczny« oznacza, że ktoś może stracić życie.”

— specjalista ds. BHP

W tym rozdziale zamieściliśmy przegląd technik programowania systemów wbudowanych. Poruszymy tematy związane z pisaniem programów dla gadżetów, które nie wyglądają jak typowe komputery z monitorem i klawiaturą. Skoncentrujemy się na zasadach, technikach programistycznych, narzędziach językowych i standardach pisania kodu wymaganych w pracy „blisko sprzętu”. Do najważniejszych poruszonych tu zagadnień będą należeć: zarządzanie zasobami i pamięcią, stosowanie wskaźników i tablic oraz operacje na bitach. Szczególny nacisk położymy na bezpieczeństwo użytkowania i alternatywy dla najniżej poziomowych narzędzi. Nie próbujemy opisać architektury urządzeń ani technik bezpośredniego dostępu do nich. Takie informacje można znaleźć w specjalistycznych publikacjach i podręcznikach. W ramach przykładu pokażemy implementację algorytmu szyfrowania i deszyfrowania.

25.1. Systemy wbudowane

25.2. Podstawy

25.2.1. Przewidywalność

25.2.2. Ideały

25.2.3. Życie z awarią

25.3. Zarządzanie pamięcią

25.3.1. Problemy z pamięcią wolną

25.3.2. Alternatywy dla ogólnej pamięci wolnej

25.3.3. Przykład zastosowania puli

25.3.4. Przykład użycia stosu

25.4. Adresy, wskaźniki i tablice

25.4.1. Niekontrolowane konwersje

25.4.2. Problem — złe działające interfejsy

25.4.3. Rozwiązanie — klasa interfejsu

25.4.4. Dziedziczenie a kontenery

25.5. Bity, bajty i słowa

25.5.1. Bity i operacje na bitach

25.5.2. Klasa bitset

25.5.3. Liczby ze znakiem i bez znaku

25.5.4. Manipulowanie bitami

25.5.5. Pola bitowe

25.5.6. Przykład — proste szyfrowanie

25.6. Standardy pisania kodu

25.6.1. Jaki powinien być standard kodowania

25.6.2. Przykładowe zasady


25.6.3. Prawdziwe standardy kodowania

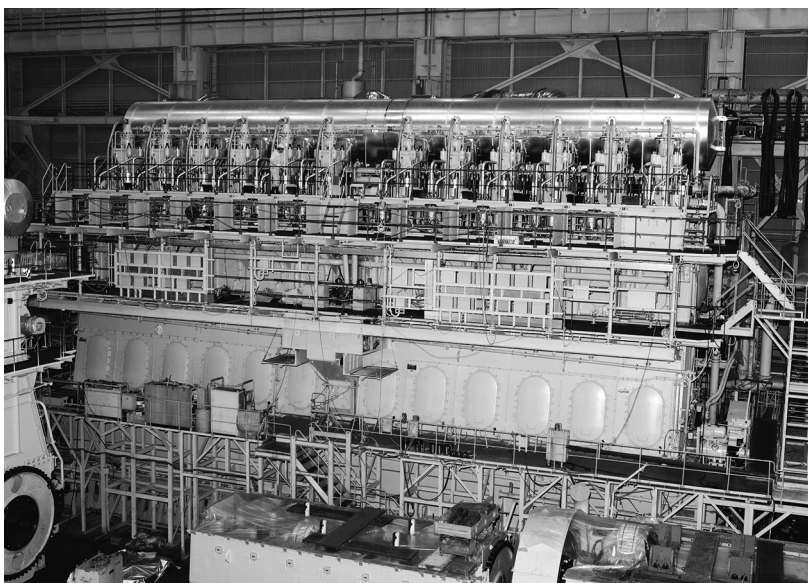
25.1. Systemy wbudowane



Większość komputerów, które istnieją na świecie, nie jest w ogóle rozpoznawana, ponieważ stanowią część większych systemów lub gadżetów. Na przykład:


- *Samochody* — w nowoczesnym samochodzie może być kilkadziesiąt komputerów: do sterowania wtryskiem paliwa, monitorowania sprawności silnika, dostrajania radia, wspomagania hamulców, kontrolowania ciśnienia gazu w oponach, sterowania wy-cieraczkami itd.
- *Telefony* — w nowoczesnym telefonie komórkowym znajdują się przynajmniej dwa komputery. Zwykle jeden z nich specjalizuje się w przetwarzaniu sygnałów.
- *Samoloty* — w nowoczesnych samolotach komputery sterują wszystkim, od systemu rozrywki pasażerów po poruszanie końcówkami skrzydeł, aby zapewnić jak najlepsze parametry lotu.
- *Aparaty cyfrowe* — istnieją aparaty fotograficzne zawierające po pięć procesorów i jeszcze po jednym dla każdej soczewki.
- *Karty kredytowe* — tzw. „inteligentne karty”.
- *Monitory i skanery medyczne* — np. skanery CAT.
- *Windy*
- *Urządzenia typu PDA*
- *Sterowniki drukarek*
- *Systemy nagłaśniające*
- *Odtwarzacze MP3*
- *Urządzenia kuchenne* — np. kuchenki i maszyny do krojenia chleba.
- *Centrale telefoniczne* — zwykle składają się z tysięcy wyspecjalizowanych komputerów.
- *Regulatory pomp* — stosowane dla pomp wody, oleju itp.
- *Roboty spawające* — są wysyłane tam, gdzie nie zmieści się człowiek lub jest dla niego zbyt niebezpiecznie.
- *Turbiny wiatrowe* — niektóre potrafią wytwarzać prąd o mocy wielu megawatów i mają wysokość nawet do 200 metrów.
- *Regulatory zapór wodnych*
- *Monitory jakości linii produkcyjnych*
- *Czytniki kodów kreskowych*
- *Roboty składające samochody*
- *Urządzenia sterujące wirówkami* — używane w wielu technikach analizy medycznej.
- *Sterowniki napędów dyskowych*

Wszystkie te komputery stanowią część większych systemów. Systemy te zwykle nie przypominają wyglądem komputerów i zwykle nikt ich tak nie traktuje. Gdy ktoś zobaczy samochód na ulicy, nie mówi do drugiej osoby „Zobacz, rozproszony system komputerowy”. Cóż, komputer w istocie jest **też** rozproszonym systemem komputerowym, ale jego działanie jest tak ściśle związane z mechanicznymi, elektronicznymi i elektrycznymi układami, że komputerów tych nie można traktować jako osobnych jednostek. Ich ograniczeń obliczeniowych (w czasie i przestrzeni) oraz samej definicji poprawności programów nie można oddzielić od reszty systemu. Często wbudowany komputer reguluje działanie jakiegoś fizycznego urządzenia i wówczas poprawność jego działania definiuje prawidłowe działanie tego urządzenia. Weźmy na przykład  duży silnik okrętowy:



Zwróć uwagę, że przy piątym cylindrze stoi człowiek. Jest to bardzo duży silnik, z tych, które napędzają największe jednostki pływające. Jeśli taki silnik ulegnie awarii, przeczytasz o tym na pierwszej stronie swojej porannej gazety. W silniku tym na każdym cylindrze znajduje się układ regulujący złożony z trzech komputerów. Każdy z tych układów ma połączenie z układem sterującym silnika (kolejne trzy komputery) za pośrednictwem dwóch niezależnych sieci. Układ sterujący silnika jest z kolei połączony ze sterownią, w której pracują inżynierowie komunikujący się z nim za pośrednictwem specjalnego GUI. Z całym systemem można też komunikować się drogą radiową (poprzez satelitę) z centrum dowodzenia linii żeglugowej. Więcej przykładów znajduje się w rozdziale 1.

Co jest więc ciekawego dla programisty w komputerze działającym jako część silnika? Mówiąc bardziej ogólnie, jakiego rodzaju problemy spotyka się przy programowaniu systemów wbudowanych, których zwykle nie ma przy pisaniu „zwykłego oprogramowania”?

- Często **najważniejsza jest niezawodność**. Awaria może mieć widowiskowe, drogie (liczone w miliardach dolarów) i potencjalnie niebezpieczne dla życia (np. ludzi znajdujących się na pokładzie statku lub zwierząt żyjących w pobliskim środowisku) skutki. 

- Często **ograniczone są zasoby (pamięć, cykle procesora, moc)** — ten rodzaj problemu jest mało prawdopodobny w przypadku komputerów w silnikach, ale może wystąpić w smartfonach, czujnikach, komputerach na pokładach próbników kosmicznych itd. W świecie, w którym królują laptopy z dwurdzeniowymi procesorami o taktowaniu 2 GHz i 8 GB pamięci RAM, najważniejszy komputer w samolocie lub próbniku kosmicznym może mieć procesor z zegarem 60 MHz i 256 KB pamięci, a mały gadżet może mieć procesor z taktowaniem poniżej 1 MHz i dysponować tylko kilkoma słowami pamięci RAM. Komputery, które są odporne na niesprzyjające warunki (wibracje, uderzenia, niestale napięcie, gorąco, zimno, wilgoć, deptanie przez robotników itd.), są zwykle dużo wolniejsze niż te, których używają studenci.
- Często **największe znaczenie ma natychmiastowe reagowanie** — jeśli układ wtryskujący paliwo pominie jeden cykl, w bardzo skomplikowanym układzie generującym 100 000 koni mechanicznych mogą zacząć dziać się straszne rzeczy. Jeśli zostanie pominiętych kilka cykli — tzn. układ wtryskowy przestanie działać na około sekundę — może zacząć dziać się coś dziwnego ze śrubami napędowymi, które mogą mieć do 10 metrów średnicy i ważyć nawet 130 ton. Naprawdę nie chciałbyś, aby to się stało.
- Często **system musi nieprzerwanie działać przez wiele lat** — dotyczy to np. systemów działających na satelitach krążących po orbicie ziemi i systemów, które są tak tanie i istnieją w tak dużej liczbie egzemplarzy, że jakiekolwiek wzmożone ilości napraw mogłyby zrujnować producenta (np. odtwarzacze MP3, karty kredytowe z chipami i układy wtrysku paliwa w samochodach). W USA szkieletowa centrala telefoniczna może nie działać przez 20 minut w ciągu 20 lat (nie ma nawet co marzyć o wyłączeniu takiego czegoś na czas zmiany oprogramowania).
- Często **bezpośrednia konserwacja jest niemożliwa lub bardzo utrudniona** — duży statek może zawinąć do portu mniej więcej raz na dwa lata, aby możliwe było przeprowadzenie konserwacji komputerów połączonej z konserwacją innych układów. Wówczas jednak muszą być dostępni odpowiedni specjaliści w odpowiednim czasie i w odpowiednim miejscu. Nieplanowane naprawy są niemożliwe do wykonania (nie mogą wystąpić żadne błędy w czasie sztormu na środku oceanu). Jeśli chodzi o próbnik krążący wokół Marsa, po prostu nie ma możliwości kogoś do niego wysłać.



Niewiele jest systemów, które dotyczą te wszystkie ograniczenia. Ale nawet jedno z nich oznacza, że pracując przy nim znajdzie odpowiedni ekspert. Nie jest naszym celem przeprowadzenie „błyskawicznego kursu na ekspertów”. Byłoby to głupie i bardzo nieodpowiedzialne z naszej strony. Chcemy zapoznać Cię z podstawowymi problemami i koncepcjami ich rozwiązań, abyś mógł zorientować się, jakie umiejętności trzeba posiadać do budowy takich systemów. Może zainteresują Cię któraś ze wspomnianych dziedzin. Ludzie projektujący i implementujący systemy wbudowane mają kluczowe znaczenie w wielu dziedzinach naszej technicyzowanej cywilizacji. W tym obszarze prawdziwy fachowiec może zrobić dużo dobrego.

Czy to dobry temat dla początkujących? Dla programistów języka C++? Tak i tak. Procesorów systemów wbudowanych jest na świecie o wiele więcej niż procesorów w komputerach osobistych. Lwia część prac programistycznych dotyczy programowania systemów wbudowanych, a więc niewykluczone, że pierwsza praca, jaką dostaniesz, będzie polegała na zaprogramowaniu właśnie takiego systemu. Ponadto widziałem osobiście, że oprogramowanie wszystkich przedstawionych do tej pory w tym rozdziale systemów zostało napisane w języku C++.

25.2. Podstawy

Znaczna część prac programistycznych związanych z programowaniem komputerów wchodzących w skład systemów wbudowanych przebiega bardzo podobnie do pisania innych rodzajów programów, dlatego ma tu zastosowanie większość tego, co napisaliśmy do tej pory. Często jednak kładzie się nacisk na coś innego — musimy dopasować swoje narzędzia językowe do ograniczeń nałożonych przez rodzaj zadania. Ponadto często zdarza się, że konieczne jest pracowanie ze sprzętem na najniższym poziomie:



- *Poprawność* — jest nawet ważniejsza niż zwykle. Nie jest to tylko abstrakcyjne pojęcie. W systemach wbudowanych poprawny program to taki, który nie tylko zwraca prawidłowe wyniki, lecz również wytwarza je w odpowiednim czasie, w odpowiedniej kolejności i przy wykorzystaniu określonej ilości zasobów. Najlepiej, gdy poprawność jest drobiazgowo określona w specyfikacji, jednak często, aby do tego dojść, trzeba poeksperymentować. Często najważniejsze testy można wykonać dopiero po zakończeniu prac nad systemem (w jego skład wchodzi komputer, na którym będzie działał program). Pełne sprecyzowanie poprawności systemu wbudowanego może być jednocześnie nieprawdopodobnie trudne i ważne. Wyrażenie „nieprawdopodobnie trudne” w tym przypadku może oznaczać „niemożliwe, jeśli weźmie się pod uwagę dostępny czas i zasoby”. Trzeba robić, co w naszej mocy, wykorzystując wszystkie dostępne narzędzia i techniki. Na szczęście zestaw specyfikacji, symulacji, testów i innych technik dostępnych w niektórych dziedzinach bywa imponujący. Wyrażenie „nieprawdopodobnie ważny” w tym przypadku oznacza: „awaria doprowadzi do uszkodzeń lub ruiny”.
- *Odporność na błędy* — trzeba bardzo skrupulatnie określić zestaw sytuacji, z którymi program powinien sobie radzić. Na przykład w przypadku zwykłego programu pisanego przez ucznia nie byłoby sprawiedliwie, gdybyśmy nagle w czasie demonstracji odłączyli zasilanie. Od zwykłych aplikacji przeznaczonych na komputery PC nie wymaga się radzenia sobie w sytuacji odcięcia dopływu energii. Jednak w systemach wbudowanych takie sytuacje mogą się zdarzać i niektóre z tych systemów muszą sobie z tym radzić. Na przykład najważniejsza część systemu może mieć dwa źródła energii, dodatkowe baterie itp. Co gorsza, w niektórych aplikacjach nie można stosować wymówek typu: „zakładałem, że sprzęt działa prawidłowo”. W długich okresach czasu i pod działaniem różnych czynników sprzęt po prostu nie działa prawidłowo. Na przykład w oprogramowaniu niektórych centrali telefonicznych i urządzeń lotniczych przewiduje się, że wcześniej czy później jakiś bit „postanowi” zmienić swoją wartość (np. z 0 na 1). Może też się zdarzyć, że jakiemuś bitowi „spodoba się” jego wartość 1 i nie zechce jej zmienić na 0 mimo „prośb”. Tego rodzaju sytuacje zdarzają się przy wystarczającej ilości pamięci, która jest używana przez wystarczająco długi czas. Proces przyspiesza wystawienie pamięci na promieniowanie, np. działające na wszystko, co znajduje się poza atmosferą Ziemi. Pracując nad jakimkolwiek systemem (wbudowanym lub nie), trzeba podjąć decyzję, jaką odporność na błędy sprzętu należy zapewnić. Zwykle przyjmuje się założenie, że sprzęt będzie działał tak, jak napisano w specyfikacji. Jeśli kwestia dotyczy systemu o wyższym stopniu ważności, założenia te mogą zostać zmodyfikowane.
- *Nieprzerwana praca* — większość systemów wbudowanych musi nieprzerwanie działać przez długi okres czasu bez żadnych zmian w oprogramowaniu, a wszelkie prace konserwacyjne muszą być przeprowadzane przez wykwalifikowanych pracowników



znających ich implementację. „Długi okres czasu” może oznaczać dni, miesiące, lata, a nawet cały cykl życiowy sprzętu. Nie dotyczy to tylko systemów wbudowanych, lecz tym się one różnią od zdecydowanej większości „zwykłych programów” i przykładowych zadań opisanych (do tej pory) w tej książce. Wymóg nieprzerwanego działania wymusza szczególne potraktowanie kwestii obsługi błędów i zarządzania zasobami. Co to jest zasób? Jest to coś, czego ilość jest ograniczona. Program uzyskuje dostęp do zasobów, wykonując określone działania („pozyskuje zasoby”, „alokuje”) i zwraca je („zwalnia”, „uwalnia”, „dealokuje”) do systemu w sposób jawny lub niejawny. Przykładami zasobów są pamięć, uchwyty do plików, połączenia sieciowe (gniazda) i blokad. Program będący częścią długo działającego systemu musi zwalniać wszystkie pozyskiwane przez siebie zasoby, z wyjątkiem tych, które są mu potrzebne cały czas. Na przykład program, który codziennie zapomina zamknąć plik, w większości systemów operacyjnych nie przetrwa dłużej niż miesiąc. Program niezwracający 100 bajtów dziennie przez rok zmarnuje ponad 32 kB — to wystarczy do unieruchomienia małego gadżetu w kilka miesięcy. Najgorsze w takich wyciekach zasobów jest to, że program będzie idealnie działał przez kilka miesięcy i nagle przestanie działać w ogóle. Jeśli program ma ulec awarii, to najlepiej, żeby to zrobił jak najwcześniej, aby można było naprawić błąd. W szczególności najważniejsze jest, aby awarie miały miejsce długo przed wysłaniem produktu do użytkowników.



- *Ograniczenia czasowe* — system wbudowany ma **ostre ograniczenia czasowe**, jeśli musi w odpowiedni sposób zareagować na coś przed upływem określonego czasu. Jeśli tak jest w większości przypadków, lecz mogą zdarzyć się od czasu do czasu opóźnienia, system ma **łagodne ograniczenia czasowe**. Przykładami systemów o łagodnych ograniczeniach czasowych są układ sterujący otwieraniem i zamykaniem okien w samochodzie i wzmacniacz w systemie nagłaśniającym. Człowiek nie jest w stanie dostrzec ułamka sekundy opóźnienia przy podnoszeniu szyby i tylko dobrze wyszkolony słuchacz potrafi wyłowić milisekundowe opóźnienie w zmianie tonacji. Przykładem systemu o ostrych ograniczeniach czasowych jest układ wtrysku paliwa, który musi trysnąć w precyzyjnie określonym czasie, gdy tłok znajduje się w określonym położeniu. Jeśli synchronizacja rozreguluje się nawet o ułamek milisekundy, spada wydajność i silnik traci moc. Duży problem z synchronizacją mógłby doprowadzić do zatrzymania silnika, co mogłoby spowodować wypadek.



- *Przewidywalność* — jest to rzecz o kluczowym znaczeniu w oprogramowaniu systemów wbudowanych. Oczywiście słowo to ma wiele znaczeń, ale w kontekście systemów wbudowanych zdefiniujemy je następująco: operacja jest **przewidywalna**, jeśli za każdym razem na tym samym komputerze zabiera tyle samo czasu oraz jeśli wszystkie tego rodzaju operacje zajmują taką samą ilość czasu. Na przykład, jeśli x i y są liczbami całkowitymi, operacja $x+y$ zawsze powinna zajmować tyle samo czasu, a $xx+yy$ zawsze powinna zajmować tyle samo czasu, jeśli xx i yy to jakieś inne liczby całkowite. Zwykle można pominąć niewielkie różnice w szybkości wykonywania spowodowane architekturą sprzętu (np. różnicami spowodowanymi przez buforowanie czy przetwarzanie potokowe) i polegać na istnieniu pewnej górnej granicy czasu wykonywania danych operacji. Operacji nieprzewidywalnych nie można używać w systemach o ostrych ograniczeniach czasowych oraz należy postępować z nimi ostrożnie w systemach działających w czasie rzeczywistym. Klasycznym przykładem nieprzewidywalnej operacji jest liniowe

przeszukiwanie listy (np. `find()`), gdy nieznana jest liczba elementów i nie można łatwo oznaczyć jej granicy. Tego rodzaju operacje można stosować w systemach o ostrych ograniczeniach czasowych tylko wówczas, gdy da się przewidzieć liczbę elementów lub przynajmniej określić jej maksymalną wartość. Aby zagwarantować odpowiedź w określonym ustalonym czasie, musimy mieć możliwość — czasami posługując się narzędziami do analizy kodu — zmierzenia czasu potrzebnego do wykonania każdej możliwej sekwencji kodu.

- *Współbieżność* — systemy wbudowane często muszą reagować na zdarzenia zewnętrzne. To stwarza zapotrzebowanie na programy, w których wiele rzeczy dzieje się na raz, ponieważ odpowiadają na realne zdarzenia, które mają miejsce jednocześnie. Program, który jednocześnie wykonuje kilka czynności, nazywa się **współbieżnym** lub **równoległym**. Niestety pasjonujące i trudne programowanie współbieżne jest poza zakresem tej książki.



25.2.1. Przewidywalność

Pod względem przewidywalności język C++ wypada całkiem nieźle, choć nie jest perfekcyjny. Wszystkie narzędzia tego języka (włącznie z wywołaniami funkcji wirtualnych) są przewidywalne, z wyjątkiem:



- alokacji pamięci wolnej za pomocą operatorów `new` i `delete` (podrozdział 25.3),
- wyjątków (podrozdział 19.5),
- operacji `dynamic_cast` (punkt A.5.7).

Wymienionych na tej liście operacji należy unikać w programach o ostrych ograniczeniach czasowych. Problemy, jakie sprawiają operatory `new` i `delete`, zostaną opisane w podrozdziale 25.3 — mają fundamentalne podłoże. Należy zauważyć też, że typ `string` z biblioteki standardowej i standardowe kontenery (`vector`, `map` itp.) pośrednio wykorzystują pamięć wolną, a więc także są nieprzewidywalne. Problem z `dynamic_cast` dotyczy aktualnych implementacji, ale nie jest fundamentalny.


Problem z wyjątkami polega na tym, że program szukający określonej klauzuli `throw` nie wie — jeśli nie przejrzy dużej ilości kodu — kiedy ją znajdzie i czy w ogóle taka istnieje. Jeśli chodzi o systemy wbudowane, lepiej żeby taka klauzula istniała, ponieważ nie można polegać na programiście, który siedzi gotowy do użycia programu do usuwania błędów. Problemy z obsługą błędów można teoretycznie rozwiązać za pomocą narzędzia informującego, która dokładnie klauzula `throw` zostanie wywołana i ile dokładnie czasu zajmie dotarcie do niej. Jednak badacze nie rozwiązyli jeszcze pewnych problemów, a więc jeśli potrzebujesz przewidywalności, musisz zastosować techniki obsługi błędów wykorzystujące kody błędów i inne przestarzałe i żmudne, ale przewidywalne techniki.

25.2.2. Ideały

Istnieje ryzyko, że wyśrubowane wymagania dotyczące wydajności i niezawodności oprogramowania systemów wbudowanych zmuszą programistę do stosowania wyłącznie niskopoziomowych narzędzi języka. Taka strategia jest możliwa do zrealizowania tylko w małych fragmentach kodu. Jednak łatwo przy tym narobić bałaganu w projekcie, trudno upewnić się,



że program jest poprawny, oraz może zwiększyć się ilość pieniędzy i czasu potrzebnych do zbudowania systemu.



Jak zwykle naszym ideałem jest pracowanie na jak najwyższym poziomie abstrakcji umożliwiającym wykonanie zadania. Nie daj się zredukować do pisania wychwalanego kodu assemblera! Jak zawsze, staraj się bezpośrednio wyrażać swoje myśli w kodzie (biorąc pod uwagę wszystkie ograniczenia). Jak zawsze, staraj się pisać jak najczystszy, jak najbardziej przejrzysty i jak najłatwiejszy w utrzymaniu kod źródłowy. Nie stosuj optymalizacji, jeśli nie musisz. Wydajność (czasowa i przestrzenna) ma często kluczowe znaczenie w systemach wbudowanych, jednak wyciskanie jej z każdego wiersza kodu jest nieporozumieniem. Ponadto dla wielu systemów wbudowanych najważniejsze są poprawność i odpowiednia szybkość. Jeśli system będzie działał szybciej niż „odpowiednio szybko”, będzie musiał beczynnienie czekać na zakończenie innych operacji. Wyciskanie z każdego wiersza kodu maksimum możliwości zabiera mnóstwo czasu, jest źródłem wielu błędów i często uniemożliwia optymalizację, ponieważ powstałe w ten sposób algorytmy i struktury danych są trudne do zrozumienia i modyfikowania. Na przykład niskopoziomowa optymalizacja często uniemożliwia optymalizację pod względem zużycia pamięci, ponieważ podobny kod pojawia się w wielu miejscach i nie może być współdzielony ze względu na mało ważne różnice.

John Bentley — znany z pisania wysokowydajnego kodu — proponuje dwa „prawa optymalizacji”:

- Pierwsze prawo: nie rób tego.
- Drugie prawo (tylko dla zaawansowanych): nie rób tego jeszcze.

Zanim zaczniesz optymalizować, upewnij się, że rozumiesz system. Dopiero dzięki jego zrozumieniu można mieć pewność, że jest — lub będzie — poprawny i niezawodny. Skoncentruj się na algorytmach i strukturach danych. Od uruchomienia pierwszej wersji skrupulatnie mierz i dostrajaj system. Na szczęście czasami zdarzają się przyjemne niespodzianki — bywa, że czysty kod działa wystarczająco szybko i nie zajmuje nadmiernej ilości pamięci. Nie można jednak na to liczyć, trzeba badać. Nieprzyjemne niespodzianki też się zdarzają.

25.2.3. Życie z awarią

Wyobraź sobie, że masz zaprojektować i zaimplementować system, który musi być niezawodny. Pod słowem „niezawodny” rozumiemy, że powinien działać bez ludzkiej interwencji przez miesiąc. Przed jakiego rodzaju awarii należy go chronić? Możemy wykluczyć kataklizmy typu wybuch słońca jako nowej lub nadeptanie systemu przez słonia. Zasadniczo jednak nigdy nie wiadomo, co może się stać. Jednak w każdym konkretnym przypadku można, a nawet trzeba, spróbować przewidzieć, jakiego rodzaju błędy są najbardziej prawdopodobne. Przykłady:

- Przepięcia lub przerwy w zasilaniu.
- Spowodowane wibracjami wypadnięcie wtyczki z gniazda.
- Uderzenie w system opadających szczątków czegoś i zniszczenie procesora.
- Upadnięcie systemu (np. od uderzenia może uszkodzić się dysk).
- Promieniowanie X może spowodować zmiany wartości bitów na takie, których nie przewiduje definicja języka.

Do najtrudniejszych do znalezienia błędów zwykle należą błędy przejściowe. **Błąd przejściowy** to taki, który występuje czasami, ale nie przy każdym uruchomieniu programu. Znamy na przykład przypadek procesora, który źle działał tylko w temperaturze powyżej 54 stopni Celsjusza. Nie przewidywano, że nagrzej się do tego stopnia. Jednak nagrzał się, gdy (niecelowo i od czasu do czasu) przykrywano go czymś w fabryce — podczas testowania w laboratorium nigdy tego nie robiono.



Błędy, które pojawiają się poza laboratorium, są najgorsze. Trudno sobie wyobrazić, ile wysiłku kosztowało zaprojektowanie i zaimplementowanie systemu umożliwiającego programistom Jet Propulsion Laboratory NASA diagnozowanie awarii sprzętu i oprogramowania marsjańskich łazików (sygnał poruszający się z prędkością światła dociera z laboratorium do urządzeń w ciągu 20 minut) oraz naprawianie problemów, gdy uda się ustalić naturę problemu.

Wiedza z danej dziedziny, tzn. dotycząca określonego systemu, jego otoczenia, sposobu jego użytkowania, jest niezbędna do zaprojektowania i zaimplementowania systemu o wysokim stopniu odporności na błędy. Tutaj opiszemy tylko pewne ogólniki. Należy zaznaczyć, że na temat każdego z nich napisano tysiące artykułów, a na ich rozwój poświęcono dziesiątki lat badań.



- *Zapobieganie wyciekom zasobów* — nie pozostawiaj wycieków. Dokładnie sprawdzaj, z jakich zasobów korzysta Twój program, i perfekcyjnie nimi zarządzaj. Nawet najmniejszy wyciek w końcu unieruchomi system lub podsystem. Do fundamentalnych zasobów zalicza się czas i pamięć. Czasami jednak programy wykorzystują też inne ich rodzaje, jak blokady, kanały komunikacyjne i pliki.
- *Replikacja* — jeśli system do funkcjonowania bezwzględnie potrzebuje jakiegoś sprzętowego zasobu (np. komputera, urządzenia wyjściowego lub koła), projektant musi zdecydować, czy w systemie tym powinno być kilka egzemplarzy tego zasobu na wszelki wypadek. Można przyjąć porażkę, jeśli sprzęt ulegnie awarii, lub dostarczyć zapasowy egzemplarz i korzystać z niego. Na przykład regulatory wtrysku paliwa w silnikach okrętowych stanowią trzy komputery połączone dwiema sieciami. Należy zauważyć, że zapasowa część nie musi być identyczna z oryginalną (np. próbnik kosmiczny może mieć silną antenę główną i słabszą zapasową). Warto też zauważyć, że zapasowa część może służyć jako wzmacniacz systemu, jeśli wszystko działa bez problemów.
- *Samokontrola* — wiedz, kiedy program (lub sprzęt) źle działa. Mogą Ci w tym pomóc niektóre urządzenia (np. nośniki pamięci), które same siebie monitorują w celu wykrycia błędów, poprawiania mniejszych błędów oraz raportowania poważnych awarii. Oprogramowanie może sprawdzać spójność swoich struktur danych, niezmienniki (punkt 9.4.3) oraz stosować wewnętrzne „testy prawdopodobieństwa” (asercje). Niestety sprawdzanie siebie samego może być zawodne. Dlatego należy zapewnić, że błędy nie wystąpią w czasie raportowania błędów — naprawę trudno dokładnie sprawdzić sprawdzanie błędów.
- *Miej szybkie wyjście ze źle działającego kodu* — dziel systemy na moduły. Obsługę błędów opieraj na modułach — każdy moduł ma do wykonania określone zadanie. Jeśli dany moduł stwierdzi, że nie może wykonać określonego zadania, może przekazać je innemu modułowi. Niech modułowe procedury obsługi błędów będą proste (dzięki czemu jest bardziej prawdopodobne, że nie będą zawierać błędów i będą wydajne) oraz niech będzie osobny moduł do radzenia sobie z poważnymi błędami. Dobry i niezawodny



system składa się z modułów i ma wiele poziomów. Poważne błędy pojawiające się na jednym poziomie są przekazywane do wyższego poziomu, aż w końcu mogą dotrzeć do człowieka. Moduł, który został powiadomiony o poważnym błędzie (takim, którego inny moduł sam nie mógł obsłużyć), może podjąć odpowiednie działania — może to być ponowne uruchomienie modułu, który wykrył ten błąd, lub przełączenie się na pracę z mniej zaawansowanym (ale solidniejszym) modułem „zapasowym”. Precyzyjna definicja modułu jest sprawą konkretnego systemu, chociaż można ogólnie powiedzieć, że jest to klasa, biblioteka, program lub wszystkie programy w komputerze.

- *Monitoruj podsystemy* — na wypadek gdyby z jakiegoś powodu same nie wykrywały problemów. W wielopoziomowym systemie wyższe poziomy mogą monitorować niższe. Wiele systemów, które naprawdę nie mogą zawieść (np. silniki okrętowe i kontrolery stacji kosmicznych), mają po trzy kopie najważniejszych podsystemów. To potrójenie nie służy tylko temu, aby mieć dwa zapasowe egzemplarze, lecz również rozwiązywaniu sporów, czy system źle działa, poprzez głosowanie 2 do 1. Stosowanie trzech egzemplarzy jest szczególnie dobrym rozwiązaniem w przypadkach, w których trudno o wielopoziomową organizację (tj. na najwyższym poziomie systemu lub podsystemu, który nie może zawieść).



Możemy poświęcić na projektowanie, ile tylko chcemy, czasu i zaimplementować go tak starannie, jak tylko potrafimy, a system i tak będzie źle działał. Przed dostarczeniem systemu użytkownikom trzeba go szczegółowo i systematycznie przetestować — rozdział 26.

25.3. Zarządzanie pamięcią

Dwa fundamentalne rodzaje zasobów komputera to czas (wykonywania instrukcji) i przestrzeń (pamięci do przechowywania danych). W języku C++ można rezerwować pamięć do przechowywania danych na trzy sposoby (podrozdział 17.4 i punkt A.4.2):

- *Pamięć statyczna* — rezerwowana przez program łączący i trwająca przez cały czas działania programu.
- *Stos (pamięć automatyczna)* — rezerwowana w chwili wywołania funkcji i zwalniana w chwili powrotu funkcji.
- *Pamięć dynamiczna (sterta)* — rezerwowana przez operator `new` i zwalniana przez `delete`.

Rozważmy te trzy rodzaje pamięci z perspektywy programisty systemów wbudowanych. W szczególności zajmiemy się zarządzaniem pamięcią pod kątem zadań, w których najważniejszą rolę odgrywa przewidywalność (punkt 25.2.1), jak programowanie systemów z ostrymi ograniczeniami czasowymi i programowanie systemów z ostrymi wymaganiami dotyczącymi bezpieczeństwa.

Pamięć statyczna nie sprawia większych kłopotów programistom systemów wbudowanych. Wszystko zostaje zrobione przed uruchomieniem programu i długo przed wdrożeniem systemu.



Pamięć stosowa może sprawiać problemy tego typu, że istnieje możliwość użycia za dużej jej ilości, ale łatwo sobie z tym poradzić. Projektant musi najczęściej udowodnić, że żadne uruchomienie programu nie spowoduje nadmiernego rozrostu stosu. Zwykle oznacza to, że musi zostać ograniczona maksymalna liczba zagnieżdżeń wywołań funkcji, czyli trzeba udowodnić,

iz łańcuch wywołań (np. funkcja `f1()` wywołuje `f2()`, która wywołuje ... `fn()`) nigdy nie stanie się za długi. W niektórych systemach spowodowało to zakaz stosowania wywołań rekurencyjnych. Takie ograniczenie w niektórych systemach i funkcjach jest uzasadnione, ale nie jest fundamentalne. Na przykład wiem, że wywołanie `factorial(10)` spowoduje najwyżej dziesięciokrotne wywołanie funkcji `factorial()`. Jednak programista systemu wbudowanego może zdecydowanie woleć iteracyjną implementację funkcji `factorial()` (podrozdział 15.5), aby uniknąć jakichkolwiek wątpliwości lub wypadków.

Rezerwacja pamięci dynamicznej jest często zabroniona lub mocno ograniczona. Tzn. stosowanie operatora `new` jest albo zabronione, albo ograniczone tylko do okresu rozruchu systemu, a operator `delete` jest zabroniony. Oto najważniejsze powody:

- *Przewidywalność* — alokacja pamięci wolnej jest nieprzewidywalna, tzn. nie ma gwarancji, że zawsze zajmie tyle samo czasu. Wielu implementacjom operatora `new` czas potrzebny na alokowanie nowego obiektu szybuje w górę, po tym jak wiele obiektów zostanie alokowanych i dealokowanych.
- *Fragmentacja* — pamięć wolna podlega fragmentacji, tzn. po alokowaniu i dealokowaniu pewnej liczby obiektów pozostała nieużywana pamięć może być pofragmentowana w tym sensie, że zawiera mnóstwo pustych luk nieużywanej i bezużytecznej przestrzeni, ponieważ luki te są zbyt małe, aby zapisać w nich obiekty takiego rodzaju, jakie są używane w aplikacji. Dlatego rozmiar przydatnej wolnej pamięci może być znacznie mniejszy niż różnica początkowej wartości i rozmiaru alokowanych obiektów.



W następnym podrozdziale wyjaśnimy, jak może dojść do takiej nieakceptowanej sytuacji. Przede wszystkim w programowaniu systemów o ostrych ograniczeniach czasowych lub wymaganiach dotyczących bezpieczeństwa należy unikać stosowania operatorów `new` i `delete`. W dalszych podrozdziałach objaśnimy systematyczne techniki unikania problemów z pamięcią wolną poprzez korzystanie ze stosów i pul.

25.3.1. Problemy z pamięcią wolną

Co takiego złego jest w operatorze `new`? W istocie problem tkwi w jednoczesnym używaniu operatorów `new` i `delete`. Rozważmy konsekwencje poniższej sekwencji alokacji i dealokacji:

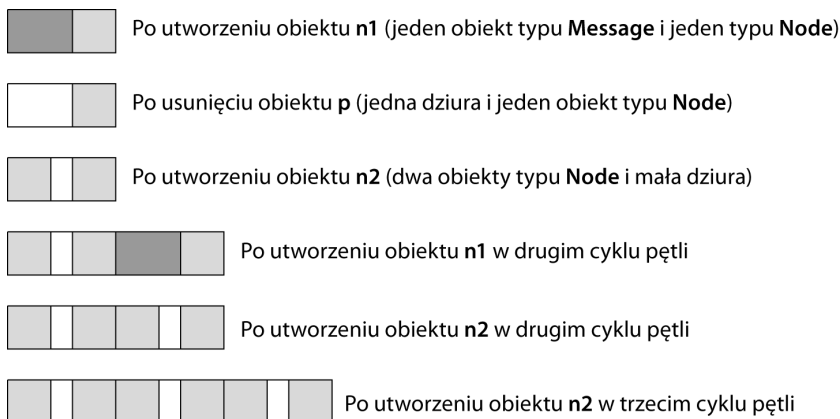
```
Message* get_input(Device&); // Tworzy obiekt typu Message w pamięci wolnej
```

```
while(/*...*/) {
    Message* p = get_input(dev);
    // ...
    Node* n1 = new Node(arg1,arg2);
    // ...
    delete p;
    Node* n2 = new Node (arg3,arg4);
    // ...
}
```

W każdym cyklu pętli tworzone są dwa obiekty typu `Node` oraz tworzymy i zaraz potem usuwamy obiekt typu `Message`. Tego rodzaju kod nie jest niczym niezwykłym w programie budującym strukturę danych na podstawie danych przychodzących od jakiegoś urządzenia. Patrząc na ten

kod, można założyć, że każdy cykl pętli spowoduje zużycie $2 * \text{sizeof}(\text{Node})$ bajtów pamięci (plus narzut pamięci wolnej). Niestety nie ma gwarancji, że zużycie pamięci będzie ograniczone do tych spodziewanych $2 * \text{sizeof}(\text{Node})$ bajtów. W istocie jest mało prawdopodobne, że tak będzie.

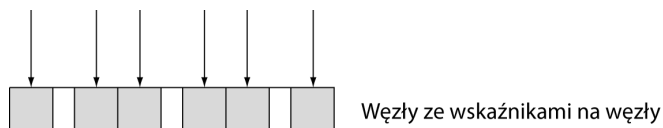
Wyobraź sobie prosty (choć wcale nie nierealny) program zarządzający pamięcią. Przyjmij też, że obiekt typu `Message` jest trochę większy od `Node`. Można graficznie przedstawić wykorzystanie pamięci wolnej, ciemnoszarym kolorem oznaczając obiekty typu `Message`, jasnoszarym obiekty typu `Node`, a białym „dziury” (tzn. pamięć nieużywaną):



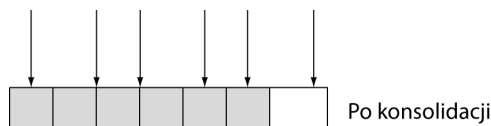
Po każdym cyklu pętli zostaje w pamięci wolnej nieużywany fragment (dziura). To może być tylko kilka bajtów, ale jeśli nie możemy ich użyć, jest to równoznaczne z wyciekaniem pamięci — a nawet mały wyciek w końcu unieruchomi długo działający program. Sytuację, w której pamięć wolna jest podzielona na zbyt małe fragmenty, aby alokować w niej obiekty, nazywa się **fragmentacją pamięci**. Program zarządzający pamięcią wolną wykorzysta wszystkie fragmenty, które są wystarczająco duże, aby coś w nich zapisać, i zostaną tylko takie, w których nic się nie zmieści. Jest to poważny problem wszystkich długo działających programów, które intensywnie wykorzystują operatory `new` i `delete`. Nierzadko zdarza się, że nie nadające się do użytku fragmenty zajmują większość pamięci. To z reguły znacznie wydłuża czas wykonywania operacji `new`, ponieważ konieczne jest przeszukanie dużej liczby obiektów i fragmentów, aby znaleźć odpowiednio duży fragment pamięci. Tego rodzaju sytuacje są niedopuszczalne w systemach wbudowanych. Problem ten może też urosnąć do wysokiej rangi w naiwnie zaprojektowanych systemach niewbudowanych.

Czemu język lub system nie może tego problemu rozwiązać? Czy nie można by było napisać programu w taki sposób, aby nie tworzył tych dziur? Najpierw przeanalizujemy najoczywistsze rozwiązanie polegające na przeniesieniu wszystkich obiektów typu `Node` w taki sposób, aby cała wolna przestrzeń zajmowała jeden ciągły obszar, co pozwoli na alokowanie większej liczby obiektów.

Niestety system nie może tego zrobić. Sęk w tym, że kod języka C++ odwołuje się bezpośrednio do obiektów w pamięci. Na przykład wskaźniki `n1` i `n2` zawierają prawdziwe adresy miejsc w pamięci. Gdyby przeniesiono znajdujące się w nich obiekty, adresy te nie wskazywałyby tego, co trzeba. Załóżmy, że przechowujemy gdzieś wskaźniki na utworzone przez siebie węzły. Interesującą nas część naszej struktury danych można przedstawić następująco:



Teraz konsolidujemy pamięć, przenosząc obiekty w taki sposób, aby cała nieużywana pamięć była w jednym kawałku:



Niestety powstał bałagan związany z tym, że przenieśliśmy obiekty w inne miejsca, a nie zaktualizowaliśmy wskazujących je wskaźników. Czemu by tego nie zrobić? Można napisać robiący to program, ale do tego potrzebna jest szczegółowa znajomość struktury danych. Zasadniczo system (system pomocniczy czasu działania C++) nie ma pojęcia, gdzie te wskaźniki są. Tzn. dla danego obiektu na pytanie: „Które wskaźniki w programie wskazują ten obiekt w tej chwili?” nie ma dobrej odpowiedzi. Nawet jeśli dałoby się ten problem łatwo rozwiązać, technika ta (zwana **usuwaniami nieużytków z konsolidacją pamięci** — ang. *compacting garbage collection*) nie zawsze jest właściwa. Aby na przykład dobrze działała, zwykle potrzebuje ponad dwa razy więcej pamięci, niż sam program by kiedykolwiek potrzebował, aby śledzić swoje wskaźniki i przenosić obiekty. Ta dodatkowa pamięć może nie być dostępna w systemie wbudowanym. Ponadto trudno jest tak skonstruować wydajny program usuwający nieużytki i konsolidujący pamięć, aby był przewidywalny.



Możemy oczywiście odpowiedzieć na pytanie: „Gdzie są wskaźniki?” dla naszych struktur danych i je skonsolidować. To by było wykonalne, ale prościej by było wcześniej uniknąć fragmentacji. W przedstawionym tu przykładzie można by było po prostu alokować oba obiekty typu `Node` przed alokowaniem wiadomości obiektu `Message`:

```
while( ... ) {
    Node* n1 = new Node;
    Node* n2 = new Node;
    Message* p = get_input(dev);
    // ... zapisanie informacji w węzłach ...
    delete p;
    // ...
}
```

Jednak zmiana struktury kodu, aby uniknąć pofragmentowania pamięci, nie jest zwykle taka łatwa. Robienie tego w niezawodny sposób w najlepszym przypadku jest bardzo trudne i często sprzeczne z innymi zasadami pisania dobrego kodu. Dlatego ogranicza się wykorzystanie pamięci wolnej do metod, które nie powodują fragmentacji. Często lepiej zapobiegać powstaniu problemów, niż je rozwiązywać.

WYPRÓBUJ



Dokończ powyższy program oraz wydrukuj adresy i rozmiary utworzonych obiektów, aby sprawdzić, czy i w jaki sposób powstają „dziury” w Twoim komputerze. Jeśli masz czas, możesz narysować podobne schematy do powyższych, aby graficznie przedstawić, co się dzieje.

25.3.2. Alternatywy dla ogólnej pamięci wolnej

Wiadomo już, że nie można dopuścić do pofragmentowania pamięci. Co z tym zrobić? Przede wszystkim należy zauważyć, że sam operator `new` nie może spowodować fragmentacji. Aby powstały dziury, potrzebny jest operator `delete`. Dlatego zabraniamy jego używania. To oznacza, że jeśli obiekt zostanie alokowany, na zawsze pozostaje częścią programu.



Czy gdyby nie było operatora `delete`, `new` byłby przewidywalny? Tzn. czy wszystkie operacje `new` zajmowałyby tyle samo czasu? W większości powszechnie wykorzystywanych implementacji tak, ale standard tego nie gwarantuje. Większość systemów wbudowanych posiada rozruchowe procedury, które przygotowują go do działania po pierwszym włączeniu lub ponownym uruchomieniu. W czasie ich wykonywania można dowolnie alokować pamięć do określonego maksimum. Można zdecydować się na wykorzystanie wówczas operatora `new`. Alternatywnie (lub dodatkowo) można sobie odłożyć globalną (statyczną) pamięć do użytku w przyszłości. Danych globalnych najlepiej unikać ze względów związanych ze strukturą programu, chociaż wykorzystanie tego mechanizmu językowego do wstępnego alokowania pamięci może być pożyteczne. Dokładny opis zasad robienia tego powinien znajdować się w standardzie kodowania systemu (podrozdział 25.6).



Dwie struktury danych są szczególnie przydatne w przewidywalnym alokowaniu pamięci:

- *Stosy* — stos to struktura danych, w której można alokować dowolną ilość pamięci (do określonego maksimum) i dealokować tylko ostatnio alokowany obiekt. Innymi słowy, stos może się powiększać i pomniejszać tylko od góry. Nie ma mowy o fragmentacji, ponieważ między żadnymi dwiema alokacjami nie może być „dziury”.
- *Pule* — pula to zbiór obiektów tego samego rozmiaru. Można je alokować i dealokować, pod warunkiem że nie alokuje się więcej obiektów, niż pula może przechowywać. Nie ma mowy o fragmentacji, ponieważ wszystkie obiekty są tego samego rozmiaru.

Zarówno w stosach, jak i pulach alokacja i dealokacja są przewidywalne i szybkie.

W związku z tym w systemach o ostrych ograniczeniach czasowych lub największym znaczeniu można definiować stosy i pule. Co ciekawsze, powinniśmy móc używać stosów i pul zdefiniowanych, zaimplementowanych i przetestowanych przez kogoś innego (jeśli ich specyfikacja odpowiada naszym potrzebom).



Należy zauważyć, że standardowe kontenery C++ (`vector`, `map` itd.) oraz standardowy typ `string` nie mogą być używane, ponieważ pośrednio korzystają z operatora `new`. Można zbudować (kupić lub pożyczyć) podobne do standardowych kontenery, które są przewidywalne, ale domyślne kontenery implementacji nie są ograniczone tylko do użytku w systemach wbudowanych.



Systemom wbudowanym zwykle stawia się bardzo surowe wymagania dotyczące niezawodności, a więc bez względu na to, jakiego rodzaju rozwiązanie zastosujemy, nie możemy pogorszyć

swojego stylu programowania, zniżając się do bezpośredniego stosowania mnóstwa narzędzi niskopoziomowych. Jest niezwykle trudno zagwarantować poprawność kodu pełnego wskaźników, jawnych konwersji itp.

25.3.3. Przykład zastosowania puli

Pula (ang. *pool*) to struktura danych, w której można alokować obiekty określonego typu i później je z niej dealokować (zwalniać). Rozmiar puli jest ograniczony, a liczbę obiektów, które może przechowywać, określa się podczas jej tworzenia. Poniżej znajduje się graficzna reprezentacja puli, jasnoszarym zaznaczono alokowane obiekty, a ciemnoszarym miejsca gotowe do alokacji:



Pulę można zdefiniować w następujący sposób:

```
template<typename T, int N>
class Pool {
public:
    Pool(); // Tworzy pulę n obiektów typu T
    T* get(); // Pobiera element T z puli. Zwraca wartość 0, jeśli nie ma wolnych T
    void free(T*); // Zwraca do puli T wydany przez funkcję get()
    int available() const; // Liczba wolnych T
private:
    // Przestrzeń dla T[N] i danych do śledzenia, które T są alokowane,
    // a które nie (np. lista wolnych obiektów)
};
```

Każdy obiekt typu Pool ma określony typ elementów i maksymalną liczbę obiektów. Można go użyć w następujący sposób:

```
Pool<Small_buffer,10> sb_pool;
Pool<Status_indicator,200> indicator_pool;

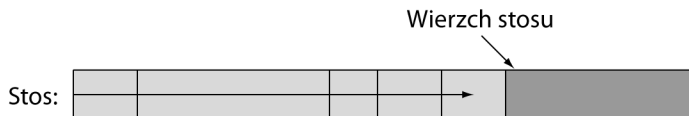
Small_buffer* p = sb_pool.get();
// ...
sb_pool.free(p);
```

Zadaniem programisty jest zagwarantować, że pula nigdy nie zostanie wyczerpana. Znaczenie słowa „zagwarantować” zależy od aplikacji. W niektórych systemach programista musi pisać kod w taki sposób, aby funkcja `get()` była wywoływana tylko, gdy jest obiekt do alokowania. W innych systemach programista może sprawdzić wynik działania tej funkcji i podjąć odpowiednie działania, jeśli jest to 0. Charakterystycznym przykładem drugiej sytuacji jest system telefoniczny zaprojektowany do obsługi najwyżej 100 000 rozmów jednocześnie. Dla każdej rozmowy jest alokowany jakiś zasób, jak np. bufor rozmów. Jeśli wyczerpią się systemowi te bufory, (np. wywołanie `dial_buffer_pool.get()` zwróci 0), system odmówi ustanowienia nowego połączenia (i może „zabić” kilka istniejących rozmów, aby odzyskać trochę sprawności). Osoba próbująca gdzieś się dodzwonić może spróbować później.

Oczywiście nasz szablon `Pool` jest tylko jedną z możliwych wersji ogólnego pojęcia puli. Na przykład w warunkach o nie tak drakońskich ograniczeniach dotyczących alokacji pamięci można definiować pule o liczbie elementów określonej w konstruktorze lub nawet takie, których liczbę elementów można później zmienić, jeśli potrzeba więcej obiektów, niż początkowo przypuszczano.

25.3.4. Przykład użycia stosu

Stos (ang. *stack*) to struktura danych, w której można alokować obiekty i dealokować tylko ostatni zaalokowany. Poniżej znajduje się graficzna reprezentacja stosu, jasnoszarym kolorem zaznaczono alokowane obiekty, a ciemnoszarym miejsce gotowe do alokacji:



Zgodnie ze wskazaniem strzałki ten stos rośnie „w prawo”.

Stos obiektów można zdefiniować tak samo jak pulę:

```
template<typename T, int N>
class Stack { // Stos N elementów typu T
    // ...
};
```

Jednak w większości systemów trzeba alokować obiekty różnych rozmiarów. Pozwala na to stos, w przeciwieństwie do puli, a więc pokażemy, jak zdefiniować stos, na którym można alokować „surową” pamięć różnych rozmiarów zamiast obiektów o stałym rozmiarze:

```
template<int N>class Stack { // Stos N bajtów
public:
    Stack();                // Tworzy stos N bajtów
    void* get(int n);        // Alokuje n bajtów na stosie
                           // Zwraca 0, jeśli nie ma wolnego miejsca
    void free();             // Zwraca na stos ostatnią wartość zwróconą przez get()
    int available() const;   // Liczba dostępnych bajtów
private:
                           // Przestrzeń dla char[N] i danych do śledzenia, co zostało alokowane,
                           // a co nie (np. wskaźnik wskazujący wierzch stosu)
};
```

Ponieważ funkcja `get()` zwraca typ `void*` wskazujący wymaganą liczbę bajtów, zadaniem programisty jest zamiana tej pamięci na obiekty potrzebnego rodzaju. Takiego stosu można użyć w następujący sposób:

```
Stack<50*1024> my_free_store; // 50 KB pamięci zostało zarezerwowanych dla stosu

void* pv1 = my_free_store.get(1024);
int* buffer = static_cast<int*>(pv1);

void* pv2 = my_free_store.get(sizeof(Connection));
Connection* pconn = new(pv2) Connection(incoming,outgoing,buffer);
```


Zastosowanie funkcji `static_cast()` opisaliśmy w podrozdziale 17.8. Konstrukcja `new(pv2)` to „operator lokujący `new`”. Oznacza: „utwórz obiekt w przestrzeni wskazywanej przez `pv2`”. Nie alokuje niczego. Przyjęto tu założenie, że typ `Connection` ma konstruktor, który przyjmie listę argumentów (`incoming, outgoing, buffer`). Jeśli nie, program nie da się skompilować.

Oczywiście nasz szablon `Stack` jest tylko jedną z możliwych wersji ogólnego pojęcia stosu. Na przykład w warunkach o nie tak drakońskich ograniczeniach dotyczących alokacji pamięci można definiować stosy o liczbie elementów określonej w konstruktorze.

25.4. Adresy, wskaźniki i tablice

Przewidywalność jest wymagana w niektórych systemach wbudowanych, niezawodność we wszystkich. Z tego powodu programiści starają się unikać narzędzi językowych i technik programistycznych, które znane są z podatności na błędy (w kontekście programowania systemów wbudowanych albo ogólnie). Głównym podejrzanym jest tu bezmyślne stosowanie wskaźników. Można wyróżnić dwa problematyczne obszary:



- Jawne (niesprawdzone i niebezpieczne) konwersje.
- Przekazywanie wskaźników do elementów tablic.

Pierwszy z wymienionych problemów można łatwo wyeliminować poprzez wniesienie surowego zakazu stosowania jawnych konwersji typów (rzutowania). Problemy ze wskaźnikami i tablicami są bardziej skomplikowane i wymagają dobrego zrozumienia. Najlepiej je rozwiązywać przy użyciu prostych klas lub narzędzi bibliotecznych (jak `array` z podrozdziału 20.9). W podrozdziale tym skupimy się zatem na rozwiązywaniu drugiego z wymienionych rodzajów problemów.

25.4.1. Niekontrolowane konwersje

Zasoby fizyczne (np. rejestry kontroli dostępu do urządzeń zewnętrznych) i ich najbardziej podstawowe sterowniki programowe zwykle w niskopoziomowych systemach przebywają pod określonymi adresami. W naszych programach musimy podawać te adresy i takim danym nadawać typy. Na przykład:

```
Device_driver* p = reinterpret_cast<Device_driver*>(0xffb8);
```

Zobacz też podrozdział 17.8. Tego rodzaju programowanie wykonuje się, mając przed oczami podręcznik lub dokumentację. Korespondencja między zasobem sprzętowym (adres rejestru lub rejestrów zasobu wyrażony zwykle jako szesnastkowa liczba całkowita) a wskaźnikami do oprogramowania, które tym zasobem manipuluje, jest krucha. Trzeba ją poprawnie ustalić bez większej pomocy ze strony kompilatora (ponieważ to nie jest kwestia dotycząca języka programowania). Zazwyczaj niezbędnym ogniwem łączącym łańcuch połączeń między aplikacją a jej zasobami sprzętowymi jest (wredna i kompletnie niekontrolowana) operacja `reinterpret_cast`.

Jeśli jawne konwersje (`reinterpret_cast`, `static_cast` itp. — punkt A.5.7) nie są niezbędne, należy ich unikać. Tego typu konwersje są znacznie rzadziej potrzebne, niż wydaje się wielu programistom z doświadczeniem w języku C lub z kodem C++ w stylu języka C.

25.4.2. Problem — źle działające interfejsy

Jak już wspominaliśmy (punkt 18.6.1), tablice często przekazuje się do funkcji jako wskaźniki na jakiś element (często pierwszy). Wówczas „tracą” one swój rozmiar, przez co odbierająca je funkcja nie wie, ile jest wskazywanych elementów. Stanowi to źródło wielu trudnych do wychwycenia i naprawienia błędów. W tym podrozdziale przeanalizujemy tego typu problemy i przedstawimy alternatywę. Zaczniemy od przykładu bardzo słabego (ale niestety nierzadko spotykanego) interfejsu i poprawimy go. Rozważmy:

```
void poor(Shape* p, int sz)           // Słaby projekt interfejsu
{
    for (int i = 0; i < sz; ++i) p[i].draw();
}

void f(Shape* q, vector<Circle>& s0) // Bardzo zły kod
{
    Polygon s1[10];
    Shape s2[10];
    // inicjalizacja
    Shape* p1 = new Rectangle(Point{0,0},Point{10,20});
    poor(&s0[0],s0.size());           // Nr 1 (przekazanie tablicy z wektora)
    poor(s1,10);                      // Nr 2
    poor(s2,20);                      // Nr 3
    poor(p1,1);                       // Nr 4
    delete p1;
    p1 = 0;
    poor(p1,1);                       // Nr 5
    poor(q,max);                      // Nr 6
}
```



Funkcja `poor()` stanowi przykład słabo zaprojektowanego interfejsu — udostępnia interfejs, który daje wywołującemu wiele możliwości do popełnienia błędu, ale nie oferuje implementującemu w zasadzie żadnej możliwości obrony przed tymi błędami.

WYPRÓBUJ



Zanim przeczytasz dalej, spróbuj znaleźć jak najwięcej błędów w funkcji `f()`. Które dokładnie wywołania funkcji `poor()` mogą spowodować awarię programu?

Na pierwszy rzut oka wywołania te wydają się nieszkodliwe, ale w rzeczywistości są dokładnie takim rodzajem kodu, który funduje programiście długie noce spędzone na szukaniu błędów, a prawdziwym specjalistom koszmary nocne.

1. Przekazanie elementu nieprawidłowego typu, np. `poor(&s0[0],s0.size())`. Także `s0` może nic nie zawierać, przez co instrukcja `&s0[0]` może być błędna.
2. Użycie „magicznej stałej” (tutaj poprawne) — `poor(s1,10)`. Także nieprawidłowy typ elementów.
3. Użycie „magicznej stałej” (tutaj niepoprawne) — `poor(s2,20)`.

4. Poprawne (łatwo to sprawdzić) — pierwsze wywołanie `poor(p1,1)`.
5. Przekazanie pustego wskaźnika — drugie wywołanie `poor(p1,1)`.
6. Może być poprawne — `poor(q,max)`. Z tego fragmentu kodu nie da się tego wywnioskować na pewno. Aby dowiedzieć się, czy `q` wskazuje tablicę zawierającą przynajmniej `max` elementów, należy znaleźć definicję `q` i `max` i sprawdzić ich wartości w interesującym nas momencie.

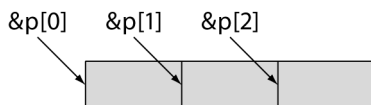
Wszystkie wymienione błędy są proste. Nie mamy tu do czynienia z żadnym subtelnym problemem algorytmicznym czy dotyczącym struktur danych. Problem polega na tym, że interfejs funkcji `poor()`, z przekazywaniem tablicy jako wskaźnika, otwiera możliwości do wystąpienia całej serii problemów. Warto zwrócić uwagę, że dzięki użyciu „technicznych” mało pomocnych nazw, takich jak `p1` i `s0`, ukryliśmy te problemy. Oczywiście zastosowanie tych krótkich, ale wprowadzających w błąd nazw jeszcze pogarsza sytuację.

Teoretycznie kompilator może znaleźć niektóre z tych błędów (np. drugie wywołanie `poor(p1,1)`, gdzie `p1==0`), ale w rzeczywistości przed katastrofą w tym przypadku uchroni nas fakt, że kompilator wykryje próbę zdefiniowania obiektów abstrakcyjnej klasy `Shape`. Nie ma to jednak związku z wadami interfejsu funkcji `poor()`, a więc niewiele by nam dało. Dalej użyjemy nieabstrakcyjnej wersji klasy `Shape`, aby nie odrywać się od problemów interfejsu.

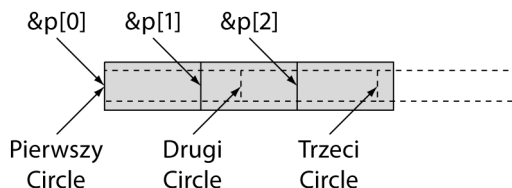
Dlaczego wywołanie `poor(&s0[0], s0.size())` jest błędne? Instrukcja `&s0[0]` odwołuje się do pierwszego elementu tablicy obiektów typu `Circle`, jest to `Circle*`. Oczekiwany jest typ `Shape*` i przekazujemy wskaźnik do obiektu klasy pochodnej od `Shape` (tutaj `Circle*`). To jest oczywiście dozwolone — musimy wykonać tę konwersję, aby pozostać w zgodzie z jedną z zasad programowania obiektowego — dostęp do różnych typów przez wspólny interfejs (tutaj `Shape`) — podrozdział 14.2. Jednak funkcja `poor()` nie używa tego `Shape*` tylko jako wskaźnika, lecz jako tablicę, przez którą przechodzi za pomocą indeksowania:

```
for (int i = 0; i<sz; ++i) p[i].draw();
```

To znaczy, że uzyskuje dostęp do obiektów, zaczynając od lokalizacji w pamięci `&p[0]`, `&p[1]`, `&p[2]` itd.:



Jeśli chodzi o adresy w pamięci, wskaźniki te znajdują się w odległości `sizeof(Shape)` jeden od drugiego (punkt 17.3.1). Na nieszczęście wywołującego funkcję `poor()` wartość `sizeof(Circle)` jest większa od `sizeof(Shape)`. Ten rozkład pamięci można przedstawić graficznie następująco:



To oznacza, że funkcja `poor()` wywołuje funkcję `draw()` ze wskaźnikiem na środek obiektów typu `Circle`! To może szybko doprowadzić do kłęski.



Wywołanie `poor(s1,10)` jest bardziej podstępne. Wykorzystuje magiczną stałą, a więc od razu kwalifikuje się jako potencjalny problem w utrzymaniu, chociaż jest jeszcze coś gorszego. Jedyny powód, dla którego użycia tablicy obiektów typu `Polygon` nie dotyka od razu ten sam problem, co tablicę obiektów typu `Circle`, jest taki, że klasa `Polygon` nie dodała do swojej klasy bazowej `Shape` żadnych danych składowych (podczas gdy `Circle` tak — podrozdziały 13.8 i 13.12). To znaczy, że `sizeof(Shape)==sizeof(Polygon)` oraz — bardziej ogólnie — obiekt typu `Polygon` ma taki sam rozkład pamięci jak `Shape`. Innymi słowy, mieliśmy po prostu szczęście. Najmniejsza zmiana definicji klasy `Polygon` spowoduje awarię. Dlatego wywołanie `poor(s1,10)` działa, ale jest bombą z opóźnionym zapłonem. Nie jest to wysokiej jakości kod.

Powyższy przykład stanowi ilustrację zasady, że to, iż „D jest B”, nie oznacza wcale, że „`Container<D>`” jest tym samym, co „`Container`” (punkt 19.3.3). Na przykład:

```
class Circle : public Shape { /*...*/ };

void fv(vector<Shape>&);
void f(Shape &);

void g(vector<Circle>& vd, Circle & d)
{
    f(d); // Dobrze: niejawna konwersja typu Circle na Shape
    fv(vd); // Błąd: nie da się konwertować typu vector<Circle> na vector<Shape>
}
```

Wiadomo już, że kod dotyczący funkcji `poor()` jest niskiej jakości, ale czy takie coś można uznać za program systemu wbudowanego? Tzn. czy należy się takim czymś przejmować w sytuacjach, gdy największe znaczenie mają bezpieczeństwo lub wydajność? Czy można to potraktować jako niebezpieczeństwo dla programistów systemów o niekluczowym znaczeniu i kazać im po prostu tego nie robić? Wiele systemów wbudowanych jest w dużym stopniu uzależnionych od GUI, które prawie zawsze jest obiektowo zorganizowane w podobny sposób do naszego przykładu. Jako przykłady można wymienić interfejs użytkownika iPodów, interfejsy niektórych telefonów komórkowych, wyświetlacze rozmaitych gadżetów, aż po samoloty. Innym przykładem jest to, że sterowniki podobnych urządzeń (jak np. rozmaitych silników elektrycznych) mogą stanowić klasyczną hierarchię klas. Innymi słowy, ten rodzaj kodu — mówiąc dokładniej tego rodzaju deklaracje funkcji — powinien nas martwić. Potrzebny jest nam bezpieczniejszy sposób na przekazywanie informacji o kolekcjach danych niepowodujący innych poważnych problemów.



Dlatego nie przekazuje się funkcji wbudowanej tablicy jako wskaźnika z liczbą określającą rozmiar. Co należy zrobić w zamian? Najprostszym rozwiązaniem jest przekazywanie referencji do kontenera, np. `vector`. Problem, z którym zetknęliśmy się w przypadku:

```
void poor(Shape* p, int sz);
```

nie może wystąpić w:

```
void general(vector<Shape>&);
```

Jeśli możesz użyć kontenera `std::vector` (lub jego odpowiednika), zawsze używaj go w interfejsach. Nigdy nie przekazuj wbudowanej tablicy jako wskaźnika z liczbą określającą rozmiar.



Jeśli nie możesz ograniczyć się do typu `vector` lub jemu równoważnych, poruszasz się po trudniejszym obszarze, na którym konieczne będzie zastosowanie trudniejszych technik i narzędzi językowych — mimo że zastosowanie dostarczonej przez nas klasy (`Array_ref`) jest proste.

25.4.3. Rozwiązanie — klasa interfejsu

Niestety w wielu systemach wbudowanych nie można korzystać z typu `vector`, ponieważ wykorzystuje on pamięć wolną. Można ten problem rozwiązać poprzez napisanie specjalnej implementacji wektora lub (co łatwiejsze) użyć kontenera, który zachowuje się jak `vector`, ale nie zarządza pamięcią. Zanim przedstawimy zarys takiej klasy interfejsowej, przemyślimy, jakie wymagania powinna spełniać:



- Stanowi referencję do obiektów w pamięci (nie posiada, nie alokuje, nie usuwa itd. obiektów).
- Zna swój rozmiar (dzięki czemu można potencjalnie zastosować sprawdzanie zakresu).
- Dokładnie zna typ swoich elementów (dzięki czemu nie może stać się źródłem błędów typów).
- Jej przekazywanie (kopiowanie) jest tak samo mało kosztowne jak przekazanie pary (wskaźnik, licznik).
- **Nie** zamienia się niejawnie we wskaźnik.
- Można z łatwością zdefiniować podzakres zakresu elementów opisanego przez obiekt interfejsu.
- Jest tak samo łatwa w użyciu, jak wbudowane tablice.

Możemy tylko zbliżyć się do spełnienia wymogu: „tak samo łatwa w użyciu, jak wbudowane tablice”. Nie chcemy, aby była tak łatwa w użyciu, że pojawi się możliwość powstawania błędów.

Poniżej znajduje się przykładowy kod takiej klasy:

```
template<typename T>
class Array_ref {
public:
    Array_ref(T* pp, int s) :p(pp), sz(s) { }

    T& operator[ ](int n) { return p[n]; }
    const T& operator[ ](int n) const { return p[n]; }

    bool assign(Array_ref a)
    {
        if (a.sz!=sz) return false;
        for (int i=0; i<sz; ++i) { p[i]=a.p[i]; }
        return true;
    }

    void reset(Array_ref a) { reset(a.p,a.sz); }
    void reset(T* pp, int s) { p=pp; sz=s; }
```


```

    int size() const { return sz; }
    // Domyślne operacje kopiowania:
    // Array_ref nie ma żadnych zasobów
    // Array_ref ma semantykę referencyjną
private:
    T* p;
    int sz;
};

```

Powyższa klasa `Array_ref` jest bardzo bliska minimalnemu rozwiązaniu:

- Brak funkcji `push_back()` (to wymagałoby użycia pamięci wolnej) i `at()` (to by wymagało zastosowania wyjątków).
- `Array_ref` jest rodzajem referencji, a więc kopiowanie oznacza zwykle skopiowanie pary `(p,sz)`.
- Dzięki podawaniu przy inicjalizacji różnych tablic można uzyskać obiekty typu `Array_ref` o tym samym typie, ale różnych rozmiarach.
- Poprzez aktualizację `(p,size)` przy użyciu funkcji `reset()` można zmienić rozmiar istniejącego obiektu typu `Array_ref` (wiele algorytmów wymaga określania podzakresów).
- Brak interfejsu iteratorów (ale można go łatwo dodać w razie potrzeby). W istocie typ `Array_ref` jest koncepcyjnie bardzo podobny do zdefiniowanego przez dwa iteratorzy zakresu.

 Obiekt typu `Array_ref` nie jest posiadaczem swoich elementów i nie zarządza pamięcią. Jest prostym mechanizmem do uzyskiwania dostępu do sekwencji elementów i przekazywania ich. Pod tym względem różni się od standardowego typu `array` (podrozdział 20.9).

Aby ułatwić tworzenie obiektów typu `Array_ref`, utworzyliśmy kilka przydatnych funkcji pomocniczych:

```

template <typename T> Array_ref<T> make_ref(T* pp, int s)
{
    return (pp) ? Array_ref<T>{pp,s} : Array_ref<T>{nullptr,0};
}

```

Jeśli obiekt typu `Array_ref` jest inicjalizowany wskaźnikiem, musi zostać jawnie podany także rozmiar. Jest to oczywiście słaby punkt tego projektu, ponieważ stwarza możliwość podania nieprawidłowej wartości. Ponadto umożliwia użycie wskaźnika będącego wynikiem niejawnej konwersji tablicy klasy pochodnej na wskaźnik na klasę bazową, jak `Polygon[10]` na `Shape*` (pierwotny straszny problem z punktu 25.4.2), ale czasami po prostu trzeba zaufać programiście.

Postanowiliśmy ostrożnie korzystać z pustych wskaźników (ponieważ są częstym źródłem problemów) i taką samą strategię stosujemy wobec pustych wektorów:

```

template <typename T> Array_ref<T> make_ref(vector<T>& v)
{
    return (v.size()) ? Array_ref<T>{&v[0],v.size()} : Array_ref<T>{nullptr,0};
}

```

Pomysł polega na przekazaniu tablicy elementów wektora. Zajmujemy się tu typem `vector`, mimo że często nie nadaje się on do użytku w tych rodzajach systemów, w których może być przydatny typ `Array_ref`. Powodem tego jest fakt, iż dzieli on kluczowe własności z kontenerami, których można tam używać (np. opartymi na pulach — punkt 25.3.3).

Na koniec zajmujemy się wbudowanymi tablicami, gdy kompilator zna rozmiar:

```
template <typename T, int s> Array_ref<T> make_ref(T (&pp)[s])
{
    return Array_ref<T>(pp,s);
}
```

Ciekawa instrukcja `T(&pp)[s]` deklaruje argument `pp` jako referencję do tablicy `s` elementów typu `T`. To pozwala na zainicjowanie obiektu typu `Array_ref` tablicą, zapamiętując jej rozmiar. Nie można zadeklarować pustej tablicy, a więc nie jest potrzebny test zera elementów:

```
Polygon ar[0]; // Błąd: brak elementów
```

Mając typ `Array_ref`, możemy jeszcze raz napisać nasz przykład:

```
void better(Array_ref<Shape> a)
{
    for (int i = 0; i<a.size(); ++i) a[i].draw();
}

void f(Shape* q, vector<Circle>& s0)
{
    Polygon s1[10];
    Shape s2[20];
    // inicjalizacja
    Shape* p1 = new Rectangle{Point{0,0},Point{10,20}};
    better(make_ref(s0)); // Błąd: wymagany Array_ref<Shape>
    better(make_ref(s1)); // Błąd: wymagany Array_ref<Shape>
    better(make_ref(s2)); // Dobrze (nie potrzeba konwersji)
    better(make_ref(p1,1)); // Dobrze: jeden element
    delete p1;
    p1 = 0;
    better(make_ref(p1,1)); // Dobrze: brak elementów
    better(make_ref(q,max)); // Dobrze (jeśli max jest w porządku)
}
```

Widać następujące cechy świadczące o poprawie:

- Kod jest prostszy. Programista rzadko musi myśleć o rozmiarach, a jeśli już musi, to są w określonym miejscu (tworzenie obiektu typu `Array_ref`), a nie porozrzucane po całym kodzie.
- Został załatwiony problem typów związany z konwersjami `Circle[]` na `Shape[]` i `Polygon[]` na `Shape[]`.
- Niejawnie poradzono sobie z problemami dotyczącymi nieprawidłowej liczby elementów dla `s1` i `s2`.

- Potencjalny problem z `max` (i innymi licznikami elementów wskaźników) stał się bardziej widoczny — jest to jedyne miejsce, w którym trzeba bezpośrednio określić rozmiar.
- Niejawnie i systematycznie obsługujemy puste wskaźniki i wektory.

25.4.4. Dziedziczenie a kontenery

Co by było, gdybyśmy chcieli potraktować kolekcję obiektów typu `Circle` jako kolekcję obiektów typu `Shape`, tzn. gdybyśmy chcieli, aby funkcja `better()` (wersja funkcji `draw_all()` — punkty 19.3.2 i 22.1.3) obsługiwała polimorfizm? Cóż, zasadniczo nie da się tego zrobić. W punktach 19.3.3 i 25.4.2 przedstawiliśmy bardzo dobre powody tego, aby system typów odmawiał akceptacji kontenerów `vector<Circle>` jako `vector<Shape>`. Z tego samego powodu nie przyjmie kontenera `Array_ref<Circle>` jako `Array_ref<Shape>`. Jeśli nie pamiętasz dlaczego, warto jeszcze raz przeczytać punkt 19.3.3, ponieważ powód ten ma fundamentalne podłoże, mimo że może to być niewygodne.



Ponadto, aby zachować polimorfizm czasu wykonywania, musimy manipulować naszymi polimorficznymi obiektami poprzez wskaźniki (albo referencje) — kropka w `a[i].draw` w funkcji `better()` mówiła sama za siebie. Powinniśmy byli spodziewać się kłopotów z polimorfizmem, jak tylko zobaczyliśmy tę kropkę zamiast strzałki (`->`).

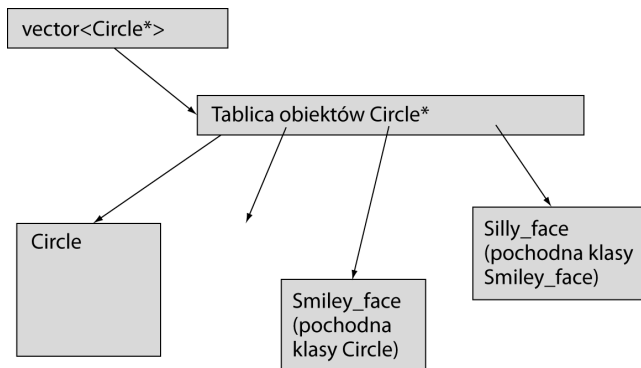
Co w takim razie możemy zrobić? Przede wszystkim **musimy** użyć wskaźników (lub referencji) zamiast bezpośrednio obiektów, a więc spróbujemy użyć `Array_ref<Circle*>`, `Array_ref<Shape*>` itd. zamiast `Array_ref<Circle>`, `Array_ref<Shape>` itd.

Nadal jednak nie możemy przekonwertować typu `Array_ref<Circle*>` na `Array_ref<Shape*>`, ponieważ moglibyśmy później wstawiać do `Array_ref<Shape*>` elementy niebędące typu `Circle*`. Jest jednak pewne sprytne wyjście:

- W tym przypadku nie chcemy modyfikować naszego obiektu typu `Array_ref<Shape*>`, chcemy tylko narysować figury! Jest to ciekawy i przydatny specjalny przypadek — nasz argument przeciwko konwersji typu `Array_ref<Circle*>` na `Array_ref<Shape*>` jest tu nieważny, jeśli obiekt typu `Array_ref<Shape*>` nie będzie modyfikowany.
- Wszystkie tablice wskaźników mają taki sam układ (bez względu na rodzaj wskazywanych obiektów), dzięki czemu nie ma problemu z układem z punktu 25.4.2.



To znaczy, że nie ma nic złego w potraktowaniu typu `Array_ref<Circle*>` jako niedającego się modyfikować typu `Array_ref<Shape*>`. Musimy więc znaleźć tylko sposób na potraktowanie `Array_ref<Circle*>` jako niemodyfikowalnego `Array_ref<Shape*>`. Rozważmy:



Nie istnieje żadna logiczna przeszkoda, aby traktować tablicę obiektów typu `Circle*` jako niedającą się modyfikować tablicę obiektów typu `Shape*` (z `Array_ref`).

Wydaje się, że zniósło nas na zaawansowane zagadnienia. Istotnie problem ten jest trudny do rozwiązania i nie da się go rozwiązać za pomocą opisanych do tej pory narzędzi. Zobaczmy jednak, jak utworzyć prawie idealną alternatywę dla naszego dysfunkcjonalnego — ale niestety zbyt często spotykanego — stylu interfejsu (wskaźnik plus licznik — punkt 25.4.2). Pamiętaj — nie wkraczaj w kompetencje ekspertów tylko po to, aby udowodnić, jaki jesteś inteligentny. Zwykle lepiej jest znaleźć jakąś bibliotekę, która została zaprojektowana i przetestowana przez ekspertów.

Najpierw zmieniamy funkcję `better()` na coś, co używa wskaźników i gwarantuje, że nie będziemy musieli „bawić się” kontenerem jako argumentem:



```
void better2(const Array_ref<Shape* const> a)
{
    for (int i = 0; i<a.size(); ++i)
        if (a[i])
            a[i]-->draw();
}
```

Używamy wskaźników, a więc powinniśmy dodać test na pusty wskaźnik. Aby mieć pewność, że funkcja `better2()` nie zmodyfikuje w niepożądany sposób naszych wektorów lub tablic poprzez `Array_ref`, dodaliśmy kilka słów `const`. Pierwsze `const` zapobiega stosowaniu operacji modyfikujących, jak `assign()` i `reset()` na rzecz naszego obiektu typu `Array_ref`. Drugie `const` za znakiem `*` oznacza, że potrzebujemy stałego wskaźnika (a nie wskaźnika do stałych), tzn. nie chcemy modyfikować wskaźników będących elementami, nawet jeśli dostępne są ku temu odpowiednie środki.

Następnie musimy rozwiązać najważniejszy problem — jak zaznaczyć, że `Array_ref<Circle*>` można konwertować:

- na coś w rodzaju `Array_ref<Shape*>` (czego można użyć w funkcji `better2()`),
- tylko na niedającą się modyfikować wersję `Array_ref<Shape*>`.

Można to zrobić poprzez dodanie do `Array_ref` operatora konwersji:

```
template<typename T>
class Array_ref {
public:
    // jak poprzednio
    template<typename Q>
    operator const Array_ref<const Q>()
    {
        // Sprawdza niejawną konwersję elementów:
        static_cast<Q>(*static_cast<T*>(nullptr)); // Sprawdza konwersję elementu:
        // Rzuca Array_ref:
        return Array_ref<const Q>{reinterpret_cast<Q*>(p),sz}; // konwersja Array_ref
    }
    // jak poprzednio
};
```

Może od tego rozboleć głowa, ale zasadniczo:

- Operator rzutuje na typ `Array_ref<const Q>` dla każdego typu `Q`, po warunkiem że możliwe jest rzutowanie elementu kontenera `Array_ref<T>` na element kontenera `Array_ref<Q>` (nie używamy wyniku tego rzutowania, tylko sprawdzamy, czy można rzutować elementy tych typów).
- Tworzymy nowy obiekt typu `Array_ref<const Q>` prymitywną metodą (`reinterpret_cast`), aby uzyskać wskaźnik na potrzebny typ elementów. Takie siłowe rozwiązania często mają swoją cenę. W tym przypadku nigdy nie stosuj konwersji `Array_ref` z klasy wykorzystującej wielokrotne dziedziczenie (punkt A.12.4).
- Zwróćmy uwagę na słowo `const` w instrukcji `Array_ref<const Q>`: uniemożliwia ono skopiowanie kontenera `Array_ref<const Q>` do starego, zwykłego, dającego się modyfikować `Array_ref<Q>`.

Ostrzegaliśmy, że to zajęcie dla ekspertów, które może przyprawiać o bóle głowy. Niemniej jednak ta wersja `Array_ref` jest łatwa w użyciu (tylko jej definicja i implementacja były trudne):

```
void f(Shape* q, vector<Circle*>& s0)
{
    Polygon* s1[10];
    Shape* s2[20];
    // inicjalizacja
    Shape* p1 = new Rectangle(Point{0,0},10);
    better2(make_ref(s0));    // Dobrze: konwersja na Array_ref<Shape*const>
    better2(make_ref(s1));    // Dobrze: konwersja na Array_ref<Shape*const>
    better2(make_ref(s2));    // Dobrze (nie potrzeba konwersji)
    better2(make_ref(p1,1));  // błąd
    better2(make_ref(q,max)); // błąd
}
```

Próby użycia wskaźników kończą się błędami, ponieważ są typu `Shape*`, podczas gdy funkcja `better2()` wymaga `Array_ref<Shape*>`. Innymi słowy funkcja ta przyjmuje coś, co zawiera wskaźniki, a nie wskaźnik. Aby przekazać do tej funkcji wskaźniki, należy umieścić je w kontenerze (np. wbudowanej tablicy lub wektorze) i przekazać ten kontener. Dla pojedynczego wskaźnika można skorzystać z niezgrabnej funkcji `make_ref(&p1,1)`. Nie ma niestety żadnego rozwiązania dla tablic (zawierających więcej niż jeden element), które nie wymaga utworzenia kontenera wskaźników na obiekty.

Podsumowując, można zrekompensować słabości tablic, tworząc proste, bezpieczne, łatwe w użyciu i wydajne interfejsy. To był najważniejszy cel postawiony w tym rozdziale. Hasło: „Każdy problem rozwiązuje jakaś inna niejasność” (cytat z Davida Wheelera) zostało zaproponowane jako „pierwsze prawo informatyki”. Zgodnie z nim rozwiązaliśmy powyższy problem interfejsu.

25.5. Bity, bajty i słowa

Już poruszaliśmy zagadnienia związane z pamięcią komputerową, jak bity, bajty i słowa, ale zasadniczo nieczęsto zajmują one głowę programisty. Zamiast tego zwykle myślimy w kategoriach różnego typu obiektów, jak `double`, `string`, `Matrix` czy `Simple_window`. W tym podrozdziale przyjrzymy się poziomowi, na którym trzeba lepiej rozumieć zasadę działania pamięci.

Jeśli czujesz, że masz braki w wiedzy dotyczącej binarnego i szesnastkowego zapisu liczb całkowitych, to może być dobra okazja do zajrzenia do punktu A.2.1.1.

25.5.1. Bity i operacje na bitach

Wyobraź sobie bajt jako sekwencję 8 bitów:

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

Zwróć uwagę na zastosowaną konwencję zapisu bitów od prawej (najmniej znaczący bit) do lewej (najbardziej znaczący bit). Teraz wyobraź sobie słowo komputerowe jako sekwencję 4 bajtów:

3:	2:	1:	0:
0xff	0x10	0xde	0xad

Tu także zastosowano konwencję zapisu od prawej do lewej, tzn. od najmniej znaczącego do najbardziej znaczącego bajta. Ilustracje te przedstawiają uproszczony obraz rzeczywistości. Są komputery, w których bajt zawiera 9 bitów (choć nie widzieliśmy już takiego od dziesięciu lat) oraz takie, w których słowo składa się z dwóch bajtów (te nie należą do rzadkości). Jeśli będziesz pamiętać, aby przed wykorzystaniem ośmio- lub czterobajtowych słów sprawdzić dokumentację systemu, wszystko powinno być w porządku.

Jeśli kod ma być przenośny, należy sprawdzić poprawność swoich założeń dotyczących rozmiarów w nagłówku `<limits>` (punkt 24.2.1). W kodzie można umieścić asercje do sprawdzenia przez kompilator:

```
static_assert(4<=sizeof(int),"typ int jest za mały");
static_assert(!numeric_limits<char>::is_signed,"char to typ ze znakiem ");
```

Pierwszym argumentem `static_assert` jest wyrażenie stałe, które w założeniu powinno być prawdziwe. Jeśli nie jest, znaczy to, że asercja się nie sprawdziła, w związku z czym kompilator drukuje powiadomienie o błędzie, w którym zawiera drugi argument, łańcuch.

Jak reprezentuje się sekwencje bitów w języku C++? Odpowiedź na to pytanie zależy od tego, ile bitów jest potrzebnych i jakie operacje na nich mają być wygodne i wydajne. Jako zbiorów bitów można używać typów całkowitoliczbowych:

- `bool` — jeden bit, ale zajmuje miejsce dla całego bajta.
- `char` — 8 bitów.
- `short` — 16 bitów.
- `int` — zwykle 32 bity, ale wiele systemów wbudowanych używa 16-bitowych typów `int`.
- `long int` — 32 lub 64 bity (ale nie mniej niż ma typ `int`).
- `long long int` — 32 lub 64 bity (ale nie mniej niż ma typ `long`).

Wymienione liczby to typowe wartości, ale w różnych implementacjach mogą być różne, a więc jeśli musisz dokładnie wiedzieć, sprawdź to. Dodatkowo w bibliotece standardowej znajdują się następujące narzędzia do operowania bitami:

- `std::vector<bool>` — gdy potrzeba jest więcej niż `8*sizeof(long)` bitów.
- `std::bitset` — gdy potrzeba jest więcej niż `8*sizeof(long)` bitów.



- `std::set` — nieuporządkowana kolekcja nazwanych bitów (punkt 21.6.5).
- `Plik` — mnóstwo bitów (punkt 25.5.6).

Dodatkowo bity można reprezentować za pomocą dwóch narzędzi językowych:

- Wyliczenia (`enum`) — podrozdział 9.5.
- Pola bitowe — punkt 25.5.5.

Ta różnorodność sposobów reprezentowania bitów odzwierciedla fakt, że w pamięci komputerowej wszystko jest zbiorem bitów. Dlatego ludzie nie mogli oprzeć się pokusie opracowania rozmaitych sposobów na oglądanie i nazywanie bitów oraz operowanie na nich. Należy zauważyć, że wbudowane narzędzia obsługują ustalone liczby bitów (np. 8, 16, 32 czy 64). Dzięki temu komputer może wykonywać na nich operacje logiczne z optymalną prędkością, stosując operacje udostępniane bezpośrednio przez sprzęt. Natomiast narzędzia z biblioteki standardowej pozwalają na wykorzystanie dowolnej liczby bitów. Może to ograniczać możliwości interpretacji, ale nie przesądza o wydajności — narzędzia biblioteki standardowej mogą być, i często są, zoptymalizowane w taki sposób, aby dobrze działały, gdy zostanie wybrana taka liczba bitów, która dobrze odpowiada warunkom sprzętowym.

Najpierw przyjrzymy się typom całkowitoliczbowym. Dla nich w języku C++ są dostępne logiczne operacje bitowe, które są bezpośrednio implementowane przez sprzęt. Operacje te mają zastosowanie dla każdego bitu swoich operandów:

Operacje bitowe

	lub	Bit n w operacji $x y$ ma wartość 1, jeśli bit n w x lub bit n w y ma wartość 1.
&	i	Bit n w operacji $x&y$ ma wartość 1, jeśli bity n w x i y mają wartości 1.
^	lub wykluczające	Bit n w operacji x^y ma wartość 1, jeśli bit n w x lub bit n w y ma wartość 1, ale nie oba na raz.
<<	przesunięcie w lewo	Bit n w operacji $x<<s$ jest bitem $n+s$ w x .
>>	przesunięcie w prawo	Bit n w operacji $x>>s$ jest bitem $n-s$ w x .
~	dopełnienie	Bit n w operacji $\sim x$ jest przeciwieństwem bitu n w x .

Może się wydawać, że operację „wykluczającego lub” (^ — czasami nazywa się xor) zaliczono do fundamentalnych. Jest ona jednak niezbędna w wielu programach przetwarzających grafikę i szyfrujących.

Kompilator nie pomyli bitowego logicznego operatora << z operatorem wyjściowym, ale programiście może się to zdarzyć. Aby uniknąć nieporozumień, zapamiętaj, że operator wyjściowy pobiera jako lewy argument strumień wyjściowy, podczas gdy logiczny operator wyjściowy pobiera w tym miejscu liczbę całkowitą.

Należy zauważyć, że operator & różni się od &&, a | od || tym, że działają na pojedyncze bity swoich operandów (punkt A.5.5) i zwracają w wyniku tyle samo bitów, ile miały argumenty. Natomiast operatory && i || zwracają wartości `true` i `false`.

Spojrzymy na kilka przykładów. Wzorce bitowe zwykle przedstawia się w notacji szesnastkowej. Dla połowy bajta (4 bitów) mamy:

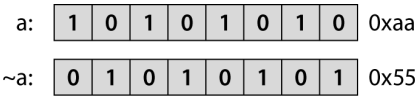
Hex	Bity	Hex	Bity
0x0	0000	0x8	1000
0x1	0001	0x9	1001
0x2	0010	0xa	1010
0x3	0011	0xb	1011
0x4	0100	0xc	1100
0x5	0101	0xd	1101
0x6	0110	0xe	1110
0x7	0111	0xf	1111

Dla liczb do 9 można by było używać wartości dziesiętnych, ale zapis szesnastkowy przypomina nam, że mamy do czynienia z wzorcami bitowymi. Notacja szesnastkowa jest bardzo przydatna w opisywaniu bajtów i słów. Bity w bajcie można wyrazić jako dwie liczby szesnastkowe. Na przykład:

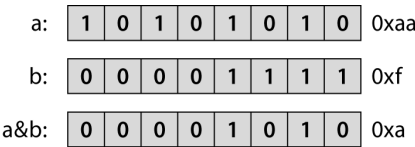
Bajt szesnastkowy	Bity
0x00	0000 0000
0x0f	0000 1111
0xf0	1111 0000
0xff	1111 1111
0xaa	1010 1010
0x55	0101 0101

Stosując słowo kluczowe unsigned (punkt 25.5.3) dla maksymalnego uproszczenia, możemy napisać:

```
unsigned char a = 0xaa;
unsigned char x0 = ~a;    // dopełnienie a
```



```
unsigned char b = 0x0f;
unsigned char x1 = a&b;    // a i b
```



```
unsigned char x2 = a^b; // wykluczające lub: a xor b
```

a:

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

 0xaa

b:

0	0	0	0	1	1	1	1
---	---	---	---	---	---	---	---

 0xf

a^b:

1	0	1	0	0	1	0	1
---	---	---	---	---	---	---	---

 0xa5

```
unsigned char x3 = a<<1; // przesunięcie o 1 w lewo
```

a:

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

 0xaa

a<<1:

0	1	0	1	0	1	0	0
---	---	---	---	---	---	---	---

 0x54

Należy zauważyć, że z za bitu numer 0 (najmniej znaczącego) zostało „wsunięte” 0, aby dopełnić bajt. Wiodące zero zwyczajnie zniknęło.

```
unsigned char x4 == a>>2; // przesunięcie o 2 w prawo
```

a:

1	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

 0xaa

a>>2:

0	0	1	0	1	0	1	0
---	---	---	---	---	---	---	---

 0x2a

Należy zaważyć, że na końcu (bit 7 — najbardziej znaczący) pojawiły się dwa zera, aby dopełnić bajt. Dwa ostatnie bity (1 i 0) zwyczajnie zniknęły.

Można sobie rysować takie wzorce bitowe, aby lepiej się zorientować, o co chodzi, ale szybko może się to stać żmudne. Poniżej znajduje się prosty program, który konwertuje liczby całkowite na ich reprezentacje bitowe:

```
int main()
{
    for (int i; cin>>i; )
        cout << dec << i << "=="
            << hex << "0x" << i << "=="
            << bitset<8*sizeof(int)>(i) << '\n';
}
```

Do wydrukowania poszczególnych bitów liczby całkowitej używamy zbioru bitów (bitset) z biblioteki standardowej:

```
bitset<8*sizeof(int)>(i)
```

bitset to ustalona liczba bitów. W tym przypadku używamy liczby bitów w typie int — 8*sizeof(int) — i inicjalizujemy nasz zbiór bitów naszą liczbą całkowitą i.

WYPRÓBUJ



Uruchom powyższy program i przepuść przez niego kilka różnych wartości, aby zorientować się, jak wyglądają binarne i szesnastkowe reprezentacje. Jeśli zaskoczą Cię reprezentacje ujemnych wartości, spróbuj jeszcze raz po przeczytaniu punktu 25.5.3.

25.5.2. Klasa `bitset`

Klasa szablonowa `bitset` z biblioteki standardowej, którą można znaleźć w nagłówku `<bitset>`, służy do reprezentowania zbiorów bitów. Każdy obiekt tej klasy ma określony rozmiar, ustalony w czasie tworzenia:

```
bitset<4> flags;
bitset<128> dword_bits;
bitset<12345> lots;
```

Obiekty klasy `bitset` są z reguły domyślnie inicjalizowane samymi zerami, chociaż zwykle podaje się im inicjalizator. Inicjalizatorem może być liczba całkowita bez znaku lub łańcuch zer i jedynek. Na przykład:

```
bitset<4> flags = 0xb;
bitset<128> dword_bits {string{"1010101010101010"}};
bitset<12345> lots;
```

W powyższym programie obiekt `lots` będzie zawierał same zera, a `dword_bits` 112 zer i 16 bitów, które zdefiniowaliśmy bezpośrednio. Jeśli w inicjalizatorze znajdują się inne znaki niż 1 i 0, zostanie zgłoszony wyjątek `std::invalid_argument`:

```
string s;
cin>>s;
bitset<12345> my_bits(s); // Może spowodować wyjątek std::invalid_argument
```

Z obiektami typu `bitset` można używać zwykłych operatorów bitowych. Przyjmijmy, że `b1`, `b2` i `b3` to obiekty typu `bitset`:

```
b1 = b2&b3; // i
b1 = b2|b3; // lub
b1 = b2^b3; // xor
b1 = ~b2;   // dopełnienie
b1 = b2<<2; // przesunięcie w lewo
b1 = b2>>3; // przesunięcie w prawo
```

Zasadniczo obiekty typu `bitset` w operacjach bitowych zachowują się jak liczby typu `int` bez znaku (punkt 25.5.3), których rozmiar został określony przez użytkownika. Wszystko, co można zrobić z liczbami `unsigned int` (z wyjątkiem działań arytmetycznych), można też zrobić z obiektami typu `bitset`. Obiekty te są w szczególności przydatne w operacjach wejścia i wyjścia:

```
cin>>b;           // Wczytuje na wejściu obiekt typu bitset
cout<<bitset<8>('c'); // Wysyła na wyjście wzorzec bitowy litery 'c'
```

Podczas wczytywania danych do obiektu typu `bitset` brane są pod uwagę tylko zera i jedynki. Rozważmy:

```
10121
```

Z powyższych danych zostanie wczytane tylko 101.

Podobnie jak w bajtach i słowach, bity w obiektach typu `bitset` są ponumerowane od prawej do lewej (od najmniej znaczącego bitu do najbardziej znaczącego bitu), a więc numeryczna wartość np. bitu numer 7 wynosi 2^7 .

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

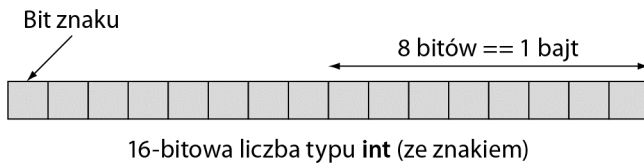
Ta kolejność bitów w obiektach typu `bitset` nie jest tylko konwencją, ponieważ obiekty te umożliwiają indeksowanie bitów. Na przykład:

```
int main()
{
    constexpr int max = 10;
    for (bitset<max> b; cin>>b; ) {
        cout << b << '\n';
        for (int i = 0; i < max; ++i) cout << b[i]; // odwrócona kolejność
        cout << '\n';
    }
}
```

Jeśli potrzebujesz więcej informacji na temat typu `bitset`, poszukaj ich w dokumentacji internetowej, podręczniku lub książce dla zaawansowanych.

25.5.3. Liczby ze znakiem i bez znaku

Język C++, jak większość języków programowania, obsługuje liczby ze znakiem i bez znaku. Liczby całkowite bez znaku są banalnie łatwe do przedstawienia w pamięci — `bit0` oznacza 1, `bit1` oznacza 2, `bit2` oznacza 4 itd., ale niestety powodują też problemy, bo jak rozpoznać, czy taka liczba jest dodatnia czy ujemna? Język C++ daje projektantom sprzętu pewną wolność, ale prawie we wszystkich implementacjach stosowana jest reprezentacja uzupełnień do dwóch. Pierwszy z lewej (najbardziej znaczący) bit pełni rolę „bitu znaku”:



Jeśli bit znaku ma wartość 1, liczba jest ujemna. Reprezentacja uzupełnienia do dwóch jest stosowana prawie wszędzie. Aby zaoszczędzić trochę papieru, rozważymy reprezentację liczb ze znakiem w czterobitowej liczbie całkowitej:

Dodatnia:	0	1	2	4	7
	0000	0001	0010	0100	0111
Ujemna:	1111	1110	1101	1011	1000
	-1	-2	-3	-5	-8

Wzorzec bitowy dla $-(x+1)$ można opisać jako dopełnienie bitów w x (znane też jako $\sim x$ — punkt 25.5.1).

Do tej pory używane były tylko liczby całkowite ze znakiem (np. `int`). Trochę lepiej stosować się do poniższych zasad:

- Używaj typów całkowitoliczbowych ze znakiem (np. `int`) do reprezentowania liczb.
- Używaj typów całkowitoliczbowych bez znaku (np. `unsigned int`) do reprezentowania zbiorów bitów.



To są całkiem dobre zasady, ale trudno się ich trzymać, ponieważ niektórzy wolą stosować liczby bez znaku w pewnych działaniach arytmetycznych, a my czasami musimy korzystać z ich kodu. W szczególności z historycznych powodów, jeszcze z początków języka C, gdy typ `int` miał 16 bitów i każdy bit się liczył, wynikiem operacji `v.size()` dla wektora jest liczba całkowita bez znaku. Na przykład:

```
vector<int> v;
// ...
for (int i = 0; i<v.size(); ++i) cout << v[i] << '\n';
```



Kompilator, który chce być „pomocny”, może ostrzec, że pomieszano wartości ze znakiem (`i`) i bez znaku (`v.size()`). Pomieszanie liczb ze znakiem i bez znaku może spowodować tragedię. Na przykład może dojść do przepełnienia zmiennej pętlowej `i`, tzn. wynik `v.size()` może być większy niż największa liczba typu `int`. Wówczas zmienna `i` mogłaby osiągnąć największą wartość, która może reprezentować dodatnią liczbę całkowitą w typie `int` ze znakiem (liczba bitów w typie `int` minus 1 do potęgi drugiej minus jeden, np. $2^{15}-1$). Następna operacja `++` nie mogłaby zostać już poprawnie wykonana i zwróciłaby wartość ujemną. Pętla nigdy by się nie skończyła! Zawsze po osiągnięciu największej liczby całkowitej zaczynalibyśmy od najmniejszej ujemnej wartości typu `int`. W przypadku typu `int`, który ma 16 bitów, taka pętla może być bardzo poważnym błędem, jeśli `v.size()` ma rozmiar $32 \cdot 1024$ lub większy. W przypadku 32-bitowych liczb typu `int` problem pojawi się, gdy `i` osiągnie wartość $2 \cdot 1024 \cdot 1024$.

W zasadzie wszystkie pokazane do tej pory pętle były niepewne i mogły spowodować błędy. Innymi słowy, programując system wbudowany, należy albo upewnić się, że pętla nigdy nie dojdzie do punktu krytycznego, albo wymienić ją na inny rodzaj pętli. Aby uniknąć tego rodzaju problemów, można użyć typu `size_type` z klasy `vector`, iteratorów lub zakresowej pętli `for`:



```
for (vector<int>::size_type i = 0; i<v.size(); ++i) cout << v[i] << '\n';
for (vector<int>::iterator p = v.begin(); p!=v.end(); ++p) cout << *p << '\n';
for (int x : v) cout << x << '\n';
```

Typ `size_type` gwarantuje, że jest bez znaku, a więc pierwsza z powyższych pętli (z liczbą całkowitą bez znaku) ma o jeden bit więcej do dyspozycji niż jeszcze wcześniejsza wersja z typem `int`. To jest już jakaś poprawa, ale dodany został tylko jeden bit (podwojenie liczby możliwych iteracji). Pętla z iteratorami nie ma takiego ograniczenia.

WYPRÓBUJ



Poniższy kod może wydawać się niewinny, ale w rzeczywistości to nieskończona pętla:

```
void infinite()
{
    unsigned char max = 160; // bardzo duża
    for (signed char i=0; i<max; ++i) cout << int(i) << '\n';
}
```

Uruchom go i wyjaśnij.



Istnieją dwa powody, dla których używa się liczb całkowitych bez znaku jako liczb całkowitych, w odróżnieniu od używania ich jako zwykłych zbiorów bitów (tj. nie używając operatorów +, -, * i /):

- Aby uzyskać dodatkowy bit precyzji.
- Aby wyrazić logiczną cechę liczby, która nie może być ujemna.

Pierwszą z tych rzeczy programiści zyskują, gdy użyją liczby bez znaku jako zmiennej pętlowej.

Problem z używaniem zarówno typów ze znakiem, jak i bez znaku polega na tym, że w języku C++ (tak jak w C) konwertują się one w zaskakujące i trudne do zapamiętania sposoby. Rozważmy:



```
unsigned int ui = -1;
int si = ui;
int si2 = ui+2;
unsigned ui2 = ui+2;
```

Ku naszemu zaskoczeniu pierwsza z powyższych inicjalizacji zakończy się powodzeniem i zmienna `ui` będzie miała wartość 4294967295, która odpowiada 32-bitowej liczbie całkowitej o takiej samej reprezentacji (wzorcu bitowym), jak liczba całkowita -1 ze znakiem (same jedynek). Niektórzy uważają to za elegancki skrót i stosują wartość -1 do oznaczania „samych jedynek”. Dla innych jest to problem. Ta sama zasada dotyczy konwersji z liczby bez znaku na liczbę ze znakiem, a więc zmienna `si` będzie miała wartość -1. Jak można było się spodziewać, zmienna `si2` będzie miała wartość 1 ($-1+2=1$), podobnie jak `ui2`. Wynik zmiennej `ui2` powinien być trochę zaskakujący — dlaczego wynikiem działania $4294967295+2$ ma być 1? Spójrz na zapis szesnastkowy liczby 4294967295 (0xffffffff) i wszystko stanie się jasne — 4294967295 jest największą 32-bitową liczbą całkowitą bez znaku, a więc nie można w 32 bitach przedstawić liczby 4294967297 — ze znakiem czy bez. Mówimy więc, że $4294967295+2$ spowodowało przepełnienie lub (dokładniej), że liczby całkowite bez znaku pozwalają na wykonywanie modułowych działań arytmetycznych, tzn. działania na 32-bitowych liczbach całkowitych są arytmetyką modularną (ang. *modular arithmetic*).



Czy wszystko jak na razie jest jasne? Nawet jeśli tak, mamy nadzieję, że przekonująco pokazaliśmy, że zabawy z dodatkowym bitem liczb całkowitych, aby uzyskać większą precyzję, to igranie z ogniem. To może być bardzo zagmatwane i stanowić źródło błędów.



Co się dzieje, gdy zostanie przekroczony zakres liczby całkowitej? Rozważmy:

```
Int i = 0;
while (++i) print(i); // Drukuje i jako liczbę całkowitą i spację
```

Jaka sekwencja wartości zostanie wydrukowana? To oczywiście zależy od definicji typu Int (nie, chociaż raz wielka litera I nie oznacza literówki). Przy użyciu typu całkowitoliczbowego o ograniczonej liczbie bitów w końcu dojdzie do przepełnienia. Jeśli typ Int jest bez znaku (np. unsigned char, unsigned int lub unsigned long long), operator ++ będzie wykonywał działania arytmetyki modularnej, a więc po największej możliwej liczbie otrzymamy 0 (i pętla zakończy działanie). Jeśli Int jest liczbą całkowitą ze znakiem (np. signed char), liczby nagle zamienią się na ujemne i zaczną od nowa zmierzać ku zeru (w którym pętla kończy działanie). Na przykład dla typu signed char zobaczymy sekwencję 1 2 ... 126 127 -128 -127... -2 -1.

Co się dzieje, gdy zostanie przekroczony zakres liczby całkowitej? Wszystko dzieje się dalej tak, jakby była wystarczająca liczba bitów, ale zostaje odrzucona ta część wyniku, która nie mieści się typie, w którym go zapisujemy. Zgodnie z tą zasadą zgubione zostaną pierwsze bity z lewej (najbardziej znaczące). Ten sam efekt można zaobserwować przy przypisywaniu:

```
int si = 257; // Nie mieści się w typie char
char c = si; // Niejawna konwersja na typ char
unsigned char uc = si;
signed char sc = si;
print(si); print(c); print(uc); print(sc); cout << '\n';
```

```
si = 129; // Nie mieści się w typie signed char
c = si;
uc = si;
sc = si;
print(si); print(c); print(uc); print(sc);
```

Otrzymujemy:

```
257    1    1    1
129   -127 129  -127
```

Wynik ten można wyjaśnić tym, że liczba 257 jest o dwa większa, niż może zmieścić się w ośmiu bitach (osiem jedynek oznacza 255), a 129 to o dwa za dużo dla 7 bitów (127 to siedem jedynek), a więc zostaje ustawiony bit znaku. Na marginesie: to pokazuje, że w naszym komputerze typ char posiada znak (c zachowuje się jak sc i inaczej niż uc).

WYPRÓBUJ

Narysuj powyższe wzorce bitowe na papierze. Później spróbuj określić wynik dla si=128. Następnie uruchom program, aby sprawdzić, czy komputer zgadza się z Twoim wynikiem.

Na marginesie: dlaczego użyliśmy funkcji print()? Moglibyśmy spróbować napisać taki kod:

```
cout << i << ' ';
```

Gdyby jednak zmienna `i` była typu `char`, na wyjściu pojawiłby się nam znak, a nie wartość całkowitoliczbowa. Aby więc wszystkie typy całkowitoliczbowe traktować jednakowo, napisaliśmy następujące definicje:

```
template<typename T> void print(T i) { cout << i << '\t'; }

void print(char i) { cout << int(i) << '\t'; }

void print(signed char i) { cout << int(i) << '\t'; }

void print(unsigned char i) { cout << int(i) << '\t'; }
```



Podsumowując, liczb całkowitych bez znaku można używać dokładnie tak samo, jak ze znakiem (włącznie ze zwykłą arytmetyką). Należy jednak tego unikać, ponieważ łatwo się pogubić i popełnić błąd.

- Staraj się nigdy nie próbować uzyskać dodatkowej precyzji poprzez użycie liczby bez znaku.
- Jeśli potrzebujesz jednego dodatkowego bitu, niedługo będziesz potrzebować kolejnego.



Niestety nie zawsze da się uniknąć arytmetyki liczb bez znaku:

- Są stosowane w indeksowaniu standardowych kontenerów.
- Niektórzy ludzie lubią arytmetykę na liczbach bez znaku.

25.5.4. Manipulowanie bitami



Po co manipuluje się bitami? Cóż, większość z nas woli tego nie robić. „Zabawy z bitami” to zajęcie na niskim poziomie, podczas którego łatwo popełnić błąd. Jeśli więc jest inne wyjście, zwykle z niego korzystamy. Bity to jednak fundament programów, który jest bardzo przydatny, a więc nie można po prostu udawać, że nie istnieje. Wypowiedź ta może mieć trochę negatywny i zniechęcający wydźwięk, ale tak właśnie miało być. Niektórzy wręcz **uwielbiają** bawić się bitami i bajtami, dlatego należy zapamiętać, że bitami należy manipulować, gdy jest to konieczne (i można dobrze się przy tym bawić), ale nie powinny one znajdować się wszędzie. Jak powiedział John Bentley: „Kto bitem wojuje, od bita zginie”.

Kiedy więc należy manipulować bitami? Czasami obiekty aplikacji są naturalnymi bitami, a więc niektóre jej działania to naturalne operacje bitowe. Jest tak w takich dziedzinach, jak wskaźniki sprzętowe (flagi), komunikacja niskopoziomowa (gdy trzeba wydobywać wartości różnych typów ze strumieni bajtów), grafika (gdy trzeba komponować obrazy z kilku poziomów obrazów) oraz szyfrowanie (przeczytaj następny podrozdział).

Pomyśl na przykład, jak wydobyć (na niskim poziomie) informacje z liczby całkowitej (może po to, aby przesłać ją w postaci bajtów, tak jak w binarnych operacjach wejścia i wyjścia):

```
void f(short val) // Przyjmuje 16-bitową czyli 2-bajtową liczbę całkowitą typu short
{
    unsigned char right = val&0xff;    // Pierwszy bajt z prawej (najmniej znaczący)
    unsigned char left = val>>8;       // Pierwszy bajt z lewej (najbardziej znaczący)
    // ...
```

```

    bool negative = val&0x8000;          // bit znaku
    // ...
}

```

Tego rodzaju operacje są często spotykane. Nazywa się je „przesunięciem i maskowaniem”. Przesuwanie (za pomocą operatorów << i >>) ma na celu przeniesienie bitów, które nas interesują do prawej (najmniej znaczącej) części słowa, gdzie łatwo się nimi manipuluje. Do maskowania służy operator & i wzorec bitowy (tu 0xff). Celem tego działania jest wyzerowanie bitów, których nie chce się w wyniku.

Do nazywania bitów zwykle wykorzystuje się wyliczenia. Na przykład:

```

enum Printer_flags {
    acknowledge=1,
    paper_empty=1<<1,
    busy=1<<2,
    out_of_black=1<<3,
    out_of_color=1<<4,
    // ...
};

```

W tej definicji każdy element ma dokładnie taką wartość, jak wskazuje jego nazwa:

out_of_color	16	0x10	0001 0000
out_of_black	8	0x8	0000 1000
busy	4	0x4	0000 0100
paper_empty	2	0x2	0000 0010
acknowledge	1	0x1	0000 0001

Przydatność takich wartości polega na tym, że można je niezależnie łączyć:

```

unsigned char x = out_of_color | out_of_black; // x otrzymuje wartość 24 (16+8)
x |= paper_empty;                             // x otrzymuje wartość 26 (24+2)

```

Zapis |= można odczytać jako: „ustaw bit” lub „ustaw kilka bitów”. Analogicznie & można przeczytać jako: „Czy bit jest ustawiony?”. Na przykład:

```

if (x& out_of_color) { // Czy bit out_of_color jest ustawiony? Tak
    // ...
}

```

Nadal można używać operatora & do maskowania:

```

unsigned char y = x &(out_of_color | out_of_black); // y otrzymuje wartość 24

```

Teraz y zawiera kopie bitów z pozycji 4 i 3 x (out_of_color i out_of_black).

Wyliczenia (enum) bardzo często są używane jako zbiory bitów. Wówczas potrzebna jest konwersja, aby wynik logicznej operacji bitowej przenieść „z powrotem” do wyliczenia. Na przykład:

```


Flags z = Printer_flags(out_of_color | out_of_black); // Rzutowanie jest konieczne

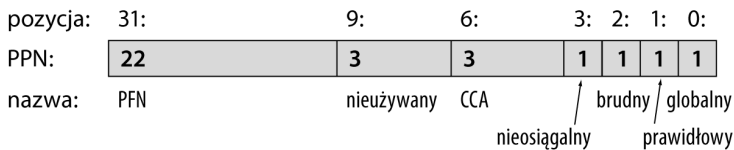
```

Rzutowanie jest potrzebne, ponieważ kompilator nie wie, że wynik działania out_of_color | out_of_black jest prawidłowy dla zmiennej Flags. Sceptycyzm kompilatora jest usprawiedliwiony

— przecież żaden element wyliczenia nie ma wartości 24 (`out_of_color|out_of_black`), ale w tym przypadku wiemy, że to przypisanie jest sensowne (kompilator natomiast tego nie wie).

25.5.5. Pola bitowe

 Jak już wspominaliśmy, interfejs sprzętowy to jedno z tych miejsc, w których często występują bity. Zwykle interfejs taki jest mieszanką bitów i liczb rozmaitych rozmiarów. Te „bity i liczby” zwykle mają nazwy i zajmują specyficzne pozycje w słowie, często nazywanym **rejestrem urządzenia** (ang. *device register*). W języku C++ istnieje specjalne narzędzie do pracy z takimi rzeczami — pola bitowe. Wyobraź sobie numer strony używany przez menedżera stron w systemie operacyjnym. Poniżej znajduje się schemat z podręcznika dotyczącego systemu operacyjnego:



Powyższe 32-bitowe słowo składa się z dwóch pól numerycznych (jedno 22-bitowe, a drugie 3-bitowe) oraz czterech flag (po jednym bicie każda). Rozmiary i pozycje tych danych są stałe. Na środku jest nawet nienazwane „pole”. Można to zaprezentować w postaci struktury:

```
struct PPN {
    // Physical Page Number R6000
    unsigned int PFN : 22 ; // Page Frame Number
    int : 3 ; // nieużywany
    unsigned int CCA : 3 ; // Cache Coherency Algorithm
    bool nonreachable : 1 ;
    bool dirty : 1 ;
    bool valid : 1 ;
    bool global : 1 ;
};
```

Musieliśmy sprawdzić w podręczniku, że PFN i CCA należy interpretować jako liczby całkowite bez znaku, chociaż moglibyśmy tę strukturę odtworzyć bezpośrednio ze schematu. Pola bitowe wypełniają słowa od lewej do prawej. Liczbę bitów podaje się w postaci całkowitoliczbowej za średnikiem. Nie można określić pozycji bezwzględnej (np. bit 8). Jeśli „zużyje się” na pola więcej bitów, niż może zmieścić się w słowie, pola, które się nie zmieszczą, zostaną przeniesione do następnego słowa. Lepiej żeby to było zamierzone. Zdefiniowanego pola bitowego używa się tak samo, jak innych zmiennych:

```
void part_of_VM_system(PPN * p )
{
    // ...
    if (p->dirty) { // zmiana zawartości
        // kopiowanie na dysk
        p->dirty = 0 ;
    }
    // ...
}
```

Pola bitowe pozwalają uzyskać dostęp do informacji w środku słowa bez przesuwania i maskowania. Na przykład mając PPN o nazwie pn, można uzyskać CCA w następujący sposób:

```
unsigned int x = pn.CCA; // Wydobywa CCA
```

Gdybyśmy do reprezentacji tych samych bitów użyli obiektu typu int o nazwie pni, moglibyśmy zamiast tego napisać:

```
unsigned int y = (pni>>4)&0x7; // Wydobywa CCA
```

To oznacza przesunięcie przestrzeni nazw w prawo, aby CCA stał się pierwszym bitem z lewej, następnie zamaskowanie wszystkich pozostałych bitów maską 0x7 (tzn. zbiorem trzech ostatnich bitów). Jeśli sprawdzisz kod maszynowy, najprawdopodobniej odkryjesz, że dla obu tych wierszy został wygenerowany taki sam kod.

Taka papka z akronimów (CCA, PPN i PFN) jest typowym zjawiskiem w kodzie tego poziomu i niewiele znaczy, gdy się ją wyjmie z kontekstu.

25.5.6. Przykład — proste szyfrowanie

Jako przykład manipulacji danymi na poziomie bitów i bajtów przedstawimy prosty algorytm szyfrowania — tzw. Tiny Encryption Algorithm (TEA). Jego pierwszą wersję napisał David Wheeler w Cambridge University (punkt 22.2.1). Mimo jego niewielkich rozmiarów świetnie radzi sobie z nieuprawnionym deszyfrowaniem.

Nie analizuj kodu zbyt uważnie (chyba że Ci na tym bardzo zależy i nie boisz się pomieszanania zmysłów). Pokazujemy ten kod tylko jako przykład realnego i przydatnego programu manipulującego na bitach. Jeśli interesuje Cię szyfrowanie, musisz sięgnąć po inną książkę. Więcej informacji i wersje tego algorytmu w różnych językach programowania można znaleźć na stronie http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm i poświęconej mu stronie profesora Simona Shepherd'a z Bradford University w Anglii. Ten kod nie ma w założeniu mówić sam za siebie (nie ma żadnych komentarzy!).

Podstawowa idea szyfrowania i deszyfrowania jest bardzo prosta. Chcę przesłać komuś tekst, ale nie chcę, żeby mógł go przeczytać ktoś inny. W związku z tym tak go modyfikuję, że staje się nieczytelny dla kogoś, kto nie wie, na czym ta modyfikacja polegała. Natomiast osoba, dla której jest przeznaczony, wie, jak odwrócić tę transformację, aby odczytać tekst. Nazywa się to szyfrowaniem. Do tego używa się algorytmów (musimy przyjąć założenie, że znanych także niepowołanym osobom) i łańcuchów nazywanych kluczami. Ja i druga osoba znamy klucz (i mamy nadzieję, że nikt niepowołany go nie zna). Gdy druga osoba odbierze zaszyfrowany tekst, deszyfruje go za pomocą klucza, innymi słowy przywraca oryginalną postać wysłanego tekstu.

Algorytm TEA przyjmuje jako argument tablicę dwóch liczb typu long bez znaku (v[0],v[1]) reprezentującą osiem znaków do zaszyfrowania, tablicę dwóch liczb typu long bez znaku (w[0],w[1]), w której zostanie zapisany zaszyfrowany tekst, oraz tablicę czterech liczb typu long bez znaku (k[0]..k[3]), która jest kluczem:

```
void encipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long * const k)
{
    static_assert(sizeof(long)==4,"Niepoprawny rozmiar słowa long dla TEA.");
```



```

unsigned long y = v[0];
unsigned long z = v[1];
unsigned long sum = 0;
const unsigned long delta = 0x9E3779B9;

for (unsigned long n = 32; n-->0; ) {
    y += (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
    sum += delta;
    z += (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
}
w[0]=y; w[1]=z;
}

```

Należy zauważyć, że wszystkie liczby są bez znaku, dzięki czemu można wykonywać na nich działania bez obawy przed niespodziankami spowodowanymi specjalnym traktowaniem ujemnych wartości. Przesunięcia (<< i >>), wykluczające lub (^) oraz bitowe i (&) wykonują niezbędną pracę przy użyciu zwykłego (bez znaku) dodawania wrzuconego na dokładkę. Ten kod jest przeznaczony dla urządzeń, w których typ long ma 4 bajty. Znajduje się w nim sporo magicznych stałych (np. przyjęto założenie, że sizeof(long) wynosi 4). Zwykle nie jest to zbyt dobrym pomysłem, ale ten konkretny fragment programu mieści się na jednej kartce. W postaci wzoru matematycznego mieści się na kopercie lub — zgodnie z pierwotnym założeniem — w głowie programisty, który ma dobrą pamięć. David Wheeler chciał móc szyfrować tekst podczas podróży bez wyciągania notatek, laptopa itp. Kod ten jest nie tylko niewielki objętościowo, lecz także szybki. Zmienna n oznacza liczbę iteracji — im większa liczba iteracji, tym silniejsze szyfrowanie. Zgodnie z naszą wiedzą szyfr TEA dla n==32 jeszcze nie został nigdy złamany.

Poniżej znajduje się kod odpowiedniej funkcji deszyfrującej:

```

void decipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long * const k)
{
    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0xC6EF3720;
    const unsigned long delta = 0x9E3779B9;
    static_assert(sizeof(long)==4, "Niepoprawny rozmiar słowa long dla TEA");
    // sum = delta<<5, ogólnie sum = delta * n
    for (unsigned long n = 32; n--> 0; ) {
        z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
        sum -= delta;
        y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
    }
    w[0]=y; w[1]=z;
}

```

Za pomocą takiego algorytmu TEA można tworzyć pliki do przesyłania informacji niezabezpieczonym łączem:

```

int main()                                // nadawca
{
    const int nchar = 2*sizeof(long);     // 64 bity

```



```

const int kchar = 2*nchar;           // 128 bitów

string op;
string key;
string infile;
string outfile;
cout << "Podaj nazwę pliku wejściowego i wyjściowego oraz klucz:\n";
cin >> infile >> outfile >> key;
while (key.size()<kchar) key += '0'; // klawisz dopełnienia
ifstream inf(infile);
ofstream outf(outfile);
if (!inf || !outf) error("Nieprawidłowa nazwa pliku.");

const unsigned long* k =
    reinterpret_cast<const unsigned long*>(key.data());

unsigned long outptr[2];
char inbuf[nchar];
unsigned long* inptr = reinterpret_cast<unsigned long*>(inbuf);
int count = 0;

while (inf.get(inbuf[count])) {
    outf << hex;           // Szesnastkowy format danych wyjściowych
    if (++count == nchar) {
        encipher(inptr,outptr,k);
        // Dopełnienie wiodącymi zerami:
        outf << setw(8) << setfill('0') << outptr[0] << ' '
            << setw(8) << setfill('0') << outptr[1] << ' ';
        count = 0;
    }
}

if (count) {               // dopełnienie
    while(count != nchar) inbuf[count++] = '0';
    encipher(inptr,outptr,k);
    outf << outptr[0] << ' ' << outptr[1] << ' ';
}
}

```

Najważniejsza w tym kodzie jest pętla `while`, reszta to tylko kod pomocniczy. Pętla ta wczytuje znaki do bufora wejściowego `inbuf` i po uzbieraniu ośmiu znaków zgodnie z wymogiem algorytmu TEA przekazuje je do funkcji `encipher()`. TEA nie dba o znaki. W istocie nie ma pojęcia, co szyfruje. Można za jego pomocą zaszyfrować np. obraz graficzny i rozmowę telefoniczną. Jedyne, co go interesuje, to otrzymanie 64 bitów (dwóch liczb typu `long` bez znaku), aby mógł wytworzyć odpowiednio zmodyfikowane 64 bity. Pobieramy więc wskaźnik na `inbuf` i rzutujemy go na typ `long*` bez znaku, który przekazujemy do TEA. To samo robimy z kluczem. TEA użyje tylko pierwszych 128 bitów (cztery liczby typu `long` bez znaku) klucza, a więc dopełniamy dane od użytkownika, aby zawsze było 128 bitów. Ostatnia instrukcja dopełnia tekst zerami, aby uzupełnić do pełnych 64 bitów (8 bajtów) wymaganych przez TEA.

Jak się przesyła zaszyfrowany tekst? W dowolny sposób, chociaż ze względu na to, że są to bity, a nie znaki ASCII czy Unicode, nie można go traktować jako zwykłego tekstu. Można rozważyć użycie binarnych operacji wejścia i wyjścia (punkt 11.3.2), ale tu zdecydowaliśmy się po prostu wyświetlić słowa w postaci liczb szesnastkowych:

```
5b8fb57c 806fbcce 2db72335 23989d1d 991206bc 0363a308
8f8111ac 38f3f2f3 9110a4bb c5e1389f 64d7efe8 ba133559
4cc00fa0 6f77e537 bde7925f f87045f0 472bad6e dd228bc3
a5686903 51cc9a61 fc19144e d3bcde62 4fdb7dc8 43d565e5
f1d3f026 b2887412 97580690 d2ea4f8b 2d8fb3b7 936cfa6d
6a13ef90 fd036721 b80035e1 7467d8d8 d32bb67e 29923fde
197d4cd6 76874951 418e8a43 e9644c2a eb10e848 ba67dcd8
7115211f dbec32069 e4e92f87 8bf3e33e b18f942c c965b87a
44489114 18d4f2bc 256da1bf c57b1788 9113c372 12662c23
eeb63c45 82499657 a8265f44 7c866aae 7c80a631 e91475e1
5991ab8b 6aedbb73 71b642c4 8d78f68b d602bfe4 d1eadde7
55f20835 1a6d3a4b 202c36b8 66ae0f2 771993f3 11d1d0ab
74a8cfd4 4ce54f5a e5fda09d acbdf110 259a1a19 b964a3a9
456fd8a3 1e78591b 07c8f5a2 101641ec d0c9d7e1 60dbeb11
b9ad8e72 ad30b839 201fc553 a34a79c4 217ca84d 30f666c6
d018e61c d1c94ea6 6ca73314 cd60def1 6e16870e 45b94dc0
d7b44cfd 96e0425a 72839f71 d5b6427c 214340f9 8745882f
0602c1a2 b437c759 ca0e3903 bd4d8460 edd0551e 31d34dd3
c3f943ed d2cae477 4d9d0b61 f647c377 0d9d303a ce1de974
f9449784 df460350 5d42b06c d4dedb54 17811b5f 4f723692
14d67edb 11da5447 67bc059a 4600f047 63e439e3 2e9d15f7
4f21bbbe 3d7c5e9b 433564f5 c3ff2597 3a1ea1df 305e2713
9421d209 2b52384f f78fbae7 d03c1f58 6832680a 207609f3
9f2c5a59 ee31f147 2ebc3651 e017d9d6 d6d60ce2 2be1f2f9
eb9de5a8 95657e30 cad37fda 7bce06f4 457daf44 eb257206
418c24a5 de687477 5c1b3155 f744fbff 26800820 92224e9d
43c03a51 d168f2d1 624c54fe 73c99473 1bce8fbb 62452495
5de382c1 1a789445 aa00178a 3e583446 dcb6d4c5 dddae173
fa168da2 60bc109e 7102ce40 9fed3a0b 44245e5d f612ed4c
b5c161f8 97ff2fc0 1dbf5674 45965600 b04c0afa b537a770
9ab9bee7 1624516c 0d3e556b 6de6eda7 d159b10e 71d5c1a6
b8bb87de 316a0fc9 62c01a3d 0a24a51f 86365842 52dabf4d
372ac18b 9a5df281 35c9f8d7 07c8f9b4 36b6d9a5 a08ae934
239efba5 5fe3fa6f 659df805 faf4c378 4c2048d6 e8bf4939
31167a93 43d17818 998ba244 55dba8ee 799e07e7 43d26aef
d5682864 05e641dc b5948ec8 03457e3f 80c934fe cc5ad4f9
0dc16bb2 a50aa1ef d62ef1cd f8fbbf67 30c17f12 718f4d9a
43295fed 561de2a0
```

WYPRÓBUJ

Klucz to bs. Odszyfruj oryginalny tekst.



Każdy specjalista od zabezpieczeń powie, że niemądrym pomysłem jest zapisywanie czystego tekstu i zaszyfrowanych plików razem oraz wypowie się na temat dopełniania, stosowania dwuliterowych kluczy itd., ale to jest książka o programowaniu, a nie o zabezpieczaniu komputerów.

Podczas testowania tych programów wczytaliśmy powyższy zaszyfrowany tekst i odzyskaliśmy oryginał. Dobrze jest zawsze przy opisanu programu mieć możliwość przeprowadzenia prostego testu poprawności.

Poniżej znajduje się najważniejsza część programu deszyfrującego:

```
unsigned long inptr[2];
char outbuf[nchar+1];
outbuf[nchar]=0; // znak oznaczający koniec
unsigned long* outptr = reinterpret_cast<unsigned long*>(outbuf);
inf.setf(ios_base::hex ,ios_base::basefield); // wejście w postaci liczb szesnastkowych

while (inf>>inptr[0]>>inptr[1]) {
    decipher(inptr,outptr,k);
    outf<<outbuf;
}
```

Zwróćmy uwagę na następujący fragment kodu:

```
inf.setf(ios_base::hex ,ios_base::basefield);
```

Odczytuje on liczby szesnastkowe. W przypadku deszyfrowania jako bity interpretujemy za pomocą rzutowania bufor wyjściowy outbuf.

Czy TEA jest przykładem programowania systemów wbudowanych? Nie bardzo, chociaż można sobie wyobrazić jego zastosowanie wszędzie tam, gdzie wymagana jest ochrona prywatności lub przeprowadzane są transakcje finansowe — to może dotyczyć także wielu „gadżetów”. Niemniej TEA demonstruje wiele cech dobrego kodu programów systemów wbudowanych — jest oparty na dobrze poznanym (matematycznym) modelu, który daje gwarancję poprawności. Jest mały, szybki i bezpośrednio wykorzystuje własności sprzętu. Styl interfejsów funkcji cipher() i decipher() nie całkiem odpowiada naszym ideałom. Należy jednak zaznaczyć, że funkcje te zostały napisane tak, aby można je było traktować jako kod w językach C i C++, przez co nie można było użyć żadnych narzędzi dostępnych tylko w języku C++. Poza tym „magiczne stałe” pochodzą z bezpośredniego ręcznego przełożenia na kod wzorów matematycznych.



25.6. Standardy pisania kodu

Istnieje wiele potencjalnych źródeł błędów. Najpoważniejsze i najtrudniejsze do poprawienia są związane z wysokopoziomowymi decyzjami dotyczącymi projektu, np. ogólną strategią obsługi wyjątków, zgodnością z określonymi standardami (lub jej brakiem), algorytmami, reprezentacją danych itd. Tego rodzaju problemy w tym podrozdziale nas *nie* interesują. Zajmiemy się błędami powstającymi z powodu pisania słabej jakości kodu, a więc takiego, w którym narzędzia językowe są niepotrzebnie używane na takie sposoby, że mogą powodować powstawanie błędów lub wyrażają takie rzeczy, które zaciemniają ich znaczenie.



Standardy kodowania mają na celu podjęcie próby rozwiązania drugiego z wymienionych rodzajów problemów poprzez zdefiniowanie „domowego stylu” prowadzącego programistów do podzbioru języka C++, który uważa się za odpowiedni dla danej dziedziny. Na przykład standard pisania kodu oprogramowania systemów wbudowanych o ostrych wymaganiach czasowych lub systemu, który musi działać „zawsze”, może zabraniać stosowania operatora new.

Zwykle też celem standardu jest zapewnienie, że kod napisany przez dwóch różnych programistów jest bardziej podobny, niż gdyby programiści ci mogli zastosować dowolne style. Może na przykład być wymóg stosowania jako pętli instrukcji `for` (co równa się z zabronieniem stosowania instrukcji `while`). W ten sposób zapewnia się większą jednolitość kodu, co może mieć znaczenie przy utrzymaniu dużych projektów. Należy pamiętać, że standardy kodowania mają na celu ulepszenie kodu programistów piszących określony rodzaj programowania. Nie ma jednego standardu kodowania odpowiedniego dla wszystkich zastosowań języka C++ i dla wszystkich programistów tego języka.



Standardy kodowania mają za zadanie rozwiązywać problemy wynikające ze sposobu wyrażania rozwiązań przez programistów, a nie z naturalnej złożoności rozwiązywanych problemów. Można powiedzieć, że standardy kodowania stanowią próbę rozwiązania nieprzewidywanych komplikacji, a nie naturalnych.

Do największych źródeł takich nieprzewidywanych komplikacji należą:



- *Nadmiernie inteligentni programiści* — używają narzędzi językowych, których nie rozumieją lub lubują się w skomplikowanych rozwiązaniach.
- *Niedouczeni programiści* — nie stosują najodpowiedniejszych narzędzi językowych i bibliotecznych.
- *Nieuzasadnione zmiany stylu programowania* — powodują, że programy wykonujące podobne zadania wyglądają inaczej i wprowadzają w błąd specjalistów od utrzymania.
- *Niewłaściwy język programowania* — prowadzi do stosowania narzędzi językowych, które są słabo przystosowane do danej dziedziny lub niezrozumiane przez określoną grupę programistów.
- *Niewystarczające wykorzystanie bibliotek* — prowadzi do dorywczego manipulowania niskopoziomowymi zasobami.
- *Nieodpowiednie standardy kodowania* — zwiększają ilość pracy lub uniemożliwiają zastosowanie najlepszych rozwiązań niektórych rodzajów problemów, tym samym stając się źródłem problemów, do których rozwiązania opracowuje się standardy kodowania.

25.6.1. Jaki powinien być standard kodowania



Dobry standard kodowania powinien pomagać programistom pisać dobry kod. To znaczy, powinien rozwiązywać małe problemy, nad którymi inaczej programista musiałby spędzić dużo czasu, próbując za każdym razem z osobna podjąć właściwą decyzję. Stare powiedzenie inżynierów głosi, że „Forma wyzwała”. Idealny standard powinien podawać gotowe przepisy instrukcyjne, co robić. Mimo iż wydaje się to oczywiste, wiele standardów kodowania to zwykle listy zakazów, które nie zawierają żadnych wskazówek na temat tego, co robić, gdy podporządkuje się im wszystkim. Samo mówienie komuś, czego ma nie robić, rzadko pomaga i tylko działa denerwująco.



Zasady dobrego stylu kodowania powinno dać się zweryfikować, najlepiej za pomocą programu. Tzn. po jego napisaniu powinno się dać z łatwością odpowiedzieć na następujące pytanie: „Czy złamałem jakąś zasadę mojego standardu kodowania?”.

Zasady dobrego standardu kodowania powinny być poparte rozsądnymi argumentami. Nie powinno się mówić programistom: „bo tak się to robi!”, ponieważ wówczas zaczynają stawiać opór. Co gorsza, zawsze próbują obalić te części standardu, które wydają się im bezcelowe lub ich zdaniem uniemożliwiają dobre wykonywanie pracy. Nie należy się spodziewać, że wszystko w danym standardzie kodowania się nam spodoba. Nawet najlepszy standard kodowania jest sumą kompromisów, a większość takich standardów zabrania niektórych praktyk, które są uważane za problematyczne — nawet jeśli Tobie nigdy nie wyrządziły żadnej szkody. Na przykład brak spójności w stosowaniu nazw jest powodem wielu nieporozumień, ale różni ludzie mają pod tym względem różne przyzwyczajenia i często bardzo nie lubią innych. Ja na przykład uważam, że identyfikatory w Notacji Wielbłądziej są „obrzydliwe” i zdecydowanie wolę styl ze znakami podkreślenia, który dla mnie jest bardziej przejrzysty i dzięki temu łatwiejszy do odczytu. Zgadza się ze mną wiele osób. Niemniej jednak jest mnóstwo rozsądnych ludzi, którzy się nie zgadzają. Oczywiście nie ma takiego standardu stosowania nazw, który podobałby się wszystkim, ale jak to zwykle bywa, stosowanie spójnego stylu jest lepsze niż niestosowanie żadnego.

Podsumowując:

- Dobrze zaprojektowany standard kodowania jest przeznaczony dla określonej dziedziny i grupy programistów.
- Dobry standard kodowania jest zarazem normatywny i restrykcyjny.
 - Często najlepszym sposobem wykorzystania normatywnych zasad jest zalecenie używania narzędzi jakiejś „fundamentalnej” biblioteki.
- Standard kodowania powinien definiować zestaw zasad dotyczących wyglądu kodu:
 - Zwykle powinien określać zasady nadawania nazw i identyfikacji, np. „Stosuj układ zalecany przez Stroustrupa”.
 - Zwykle powinien określać podzbiór języka, np. „Nie używaj instrukcji `new` i `throw`”.
 - Zwykle powinien określać zasady stosowania komentarzy, np. „Do każdej funkcji powinien być dołączony komentarz opisujący jej działanie”.
 - Często powinien wymagać stosowania określonych bibliotek, np. „Używaj narzędzi z biblioteki `<iostream>` zamiast `<stdio.h>`” lub „Używaj typów `vector` i `string` zamiast wbudowanych tablic i łańcuchów w stylu języka C”.
- Celem większości standardów kodowania jest poprawienie:
 - niezawodności,
 - przenośności,
 - łatwości utrzymania,
 - łatwości testowania,
 - możliwości wielokrotnego użycia,
 - rozszerzalności,
 - czytelności.



- Dobry standard kodowania jest lepszy niż żaden. Nie można zacząć dużego (wieloosobowego i wieloletniego) projektu bez żadnego standardu.
- Zły standard kodowania może być jeszcze gorszy niż żaden. Na przykład standardy kodowania w języku C++, które ograniczają zestaw narzędzi do czegoś w rodzaju podzbioru języka C, są szkodliwe. Niestety słabej jakości standardy kodowania nie są rzadkie.
- Programiści nie lubią żadnych standardów kodowania, nawet tych dobrych. Większość programistów woli pisać swoje programy dokładnie tak, jak lubi.

25.6.2. Przykładowe zasady

Aby pokazać, na czym mniej więcej polegają standardy kodowania, przedstawimy kilka przykładowych zasad. Oczywiście wybraliśmy takie, które naszym zdaniem mogą być przydatne. Należy jednak zaznaczyć, że nigdy nie widzieliśmy jeszcze standardu kodowania, który miał mniej niż 35 stron, a większość jest znacznie dłuższa. Dlatego tego zestawu nie można w żadnej mierze uznać za wyczerpujący. Ponadto każdy dobry standard kodowania jest zaprojektowany specjalnie dla określonej dziedziny i konkretnej grupy programistów. Nie żywimy zatem żadnych pretensji do uniwersalności.

Zasady zostały ponumerowane i każda z nich jest opatrzona krótkim uzasadnieniem. Wiele z nich zawiera przykłady pomagające lepiej je zrozumieć. Rozróżniamy **zalecenia**, które programiści mogą czasami ignorować, i **ścisłe zasady**, których należy bezwzględnie się trzymać. W realnym świecie taką zasadę można złamać tylko, gdy się otrzyma pisemne pozwolenie od zwierzchnika. Każde złamanie zalecenia lub ścisłej zasady musi zostać udokumentowane w postaci komentarza. Wszelkie wyjątki od reguł można wymienić w opisie tych reguł. Ścisłe zasady są oznaczane wielkimi literami Z i numerami, a zalecenia małymi literami z i numerami.

Oto zastosowana klasyfikacja zasad:

- Zasady ogólne,
- Preprocesor,
- Nazwy i układ,
- Funkcje i wyrażenia,
- Klasy,
- Ostre ograniczenia czasowe,
- Systemy o krytycznym znaczeniu.

Zasady „Ostre ograniczenia czasowe” i „Systemy o krytycznym znaczeniu” dotyczą tylko projektów zakwalifikowanych jako takie.

Nasz standard kodowania jest niepełny w porównaniu z realnymi standardami (nie wiadomo np., co oznacza „krytyczny”), co widać między innymi w związku zasad. Podobieństwa tego zestawu zasad do JSF++ (punkt 25.6.3) są nieprzypadkowe — pomagałem w formułowaniu zasad JSF++. Jednak przedstawione w tej książce przykłady kodu nie odpowiadają poniższemu zasadom — kod w tej książce nie jest przecież kodem krytycznego systemu wbudowanego.

Zasady ogólne

Z100: funkcje i klasy mogą składać się z nie więcej niż 200 logicznych wierszy kodu (nie licząc komentarzy).

Uzasadnienie: długie funkcje i klasy są zwykle skomplikowane, przez co trudno je zrozumieć i testować.

z101: kod każdej funkcji i klasy powinien zmieścić się na jednym ekranie monitora i rozwiązywać jedno logiczne zadanie.

Uzasadnienie: jeśli programista widzi tylko część funkcji lub klasy, istnieje większe ryzyko, że przeoczy jakiś problem. Funkcje wykonujące kilka logicznych zadań są zwykle dłuższe i bardziej skomplikowane.

Z102: kod powinien być zgodny ze standardem ISO/IEC 14882:2003(E) języka C++.

Uzasadnienie: różne wersje i rozszerzenia standardu ISO/IEC 14882 mogą być mniej stabilne, słabiej zdefiniowane oraz ograniczać przenośność.

Preprocesor

Z200: zabronione jest używanie makr poza zarządzaniem kodem źródłowym przy użyciu dyrektyw `#ifdef` i `#ifndef`.

Uzasadnienie: makra nie przestrzegają zasad zakresów i typów. Zastosowanie makra nie wynika w oczywisty sposób z tekstu programu.

Z201: za pomocą dyrektywy `#include` mogą być dołączane tylko pliki nagłówkowe (`*.h`).

Uzasadnienie: dyrektywa `#include` służy tylko do dołączania deklaracji interfejsów — nie szczegółów implementacyjnych.

Z202: wszystkie dyrektywy `#include` muszą znajdować się przed wszystkimi deklaracjami innymi niż dyrektywy preprocesora.

Uzasadnienie: dyrektywę `#include` umieszczoną w środku pliku można łatwo przeoczyć, co może stać się źródłem nieporozumień związanych z różnym interpretowaniem jednej nazwy w różnych miejscach.

Z203: pliki nagłówkowe (`*.h`) nie mogą zawierać definicji niestałych (nie `const`), zmiennych ani definicji funkcji innych niż „inline” szablonowe.

Uzasadnienie: pliki nagłówkowe powinny zawierać deklaracje interfejsów — nie szczegóły implementacyjne. Jednak stałe często wchodzi w skład interfejsów, niektóre bardzo proste funkcje muszą być „inline” (a więc w nagłówkach) ze względów wydajnościowych oraz aktualne implementacje szablonów wymagają, aby w nagłówkach znajdowały się kompletne definicje szablonów.

Nazwy i układ

Z300: w każdym pliku z kodem źródłowym muszą być jednolicie stosowane wcięcia.

Uzasadnienie: czytelność i styl.

Z301: każda instrukcja zaczyna się w nowym wierszu.

Uzasadnienie: czytelność.

Przykład:

```
int a = 7; x = a+7; f(x,9); // źle
int a = 7;                // dobrze
x = a+7;                  // dobrze
f(x,9);                   // dobrze
```

Przykład:

```
if (p<q) cout << *p;    // źle
```

Przykład:

```
if (p<q)
    cout << *p;        // dobrze
```

Z302: Nazwy identyfikatorów powinny mieć jakieś znaczenie.

Identyfikatory mogą zawierać powszechnie znane skróty i akronimy.

Takie nazwy jak *x*, *y*, *i* oraz *j* używane w konwencjonalny sposób uważane są za opisowe.

Należy stosować styl `liczba_elementów`, a nie `liczbaElementów`.

Nie należy stosować notacji węgierskiej.

Nazwy typów, szablonów i przestrzeni nazw mają zaczynać się wielką literą.

Należy unikać stosowania zbyt długich nazw.

Przykład: `Device_driver` i `Buffer_pool`.

Uzasadnienie: czytelność.

Uwaga: identyfikatory zaczynające się od znaku podkreślenia są zarezerwowane przez implementację języka, a więc ich stosowanie we własnym kodzie jest zabronione.

Wyjątek: można używać nazw z zatwierdzonych bibliotek.

Z303: identyfikatory nie mogą różnić się tylko:

- wielkością liter,
- obecnością lub brakiem znaku podkreślenia,
- obecnością litery *O* zamiast cyfry *0* lub litery *D*,
- obecnością litery *I* zamiast cyfry *1* lub litery *l*,
- obecnością litery *S* zamiast cyfry *5*,
- obecnością litery *Z* zamiast cyfry *2*,
- obecnością litery *n* zamiast litery *h*.

Przykład: `Head` i `head`

Uzasadnienie: czytelność.

Z304: identyfikatory nie mogą być pisane samymi wielkimi literami lub składać się z samych wielkich liter i znaków podkreślenia.

Przykład: `BLUE` i `BLUE_CHEESE`

Uzasadnienie: w całości wielkimi literami pisze się nazwy makr, które mogą być używane w dołączanych za pomocą dyrektywy `#include` plikach zatwierdzonych bibliotek.

Wyjątek: nazwy makr zabezpieczeń `#include`.

Funkcje i wyrażenia:

z400: identyfikatory w zakresie wewnętrznym nie mogą być identyczne z identyfikatorami w zakresie zewnętrznym.

Przykład:

```
int var = 9; { int var = 7; ++var; } // Źle: var zastania var
```

Uzasadnienie: czytelność.

Z401: Deklaracje powinny być zamieszczane w jak najmniejszym zakresie.

Uzasadnienie: dzięki utrzymywaniu struktur inicjalizujących i wykorzystujących dane jednostki łatwiej uniknąć pomyłek. Gdy zmienna wychodzi poza zakres dostępności, zajmowane przez nią zasoby są zwalniane.

Z402: zmienne muszą być inicjalizowane.

Przykład:

```
int var; // Źle — niezainicjowanie zmiennej var
```

Uzasadnienie: niezainicjowane zmienne są częstym źródłem błędów.

Wyjątek: zmienna, która jest natychmiast wypełniana danymi wejściowymi, nie musi być zainicjowana.

Uwaga: wiele typów, w tym vector i string, ma domyślny konstruktor gwarantujący inicjalizację.

Z403: zabrania się rzutowania.

Powód: rzutowanie jest częstym źródłem błędów.

Wyjątek: można stosować dynamic_cast.

Wyjątek: nazwane operacje rzutowania można stosować do konwertowania adresów na wskaźniki oraz typu void* odebranego ze źródeł zewnętrznych (np. biblioteki GUI) na wskaźniki odpowiedniego typu.

Z404: w interfejsach nie należy stosować tablic wbudowanych, tzn. wskaźnik przekazany jako argument funkcji będzie traktowany jako wskaźnik na jeden element. Do przekazywania tablic należy używać typu Array_ref.

Uzasadnienie: tablice są przekazywane jako wskaźniki, a liczba ich elementów nie jest przekazywana do wywoływanej funkcji. Ponadto kombinacje niejawnych konwersji tablic na wskaźniki oraz typów pochodnych na bazowe mogą doprowadzić do błędów pamięci.

Klasy

Z500: za pomocą słowa kluczowego class należy definiować klasy niezawierające żadnych publicznych danych składowych. Za pomocą słowa kluczowego struct należy definiować klasy niezawierające żadnych prywatnych danych składowych. Nie należy tworzyć klas zarówno z publicznymi, jak i prywatnymi danymi składowymi.

Uzasadnienie: przejrzystość.

z501: jeśli klasa ma destruktor lub składową typu wskaźnikowego lub referencyjnego, musi mieć zdefiniowane lub zabronione konstruktor kopiujący i przypisanie kopiujące.

Uzasadnienie: destruktor zwykle zwalnia jakieś zasoby. Domyślna semantyka kopiowania rzadko odpowiada składowym wskaźnikowym i referencyjnym oraz klasom z destruktorami.

Z502: jeśli klasa zawiera wirtualną funkcję, musi też mieć wirtualny destruktor.

Uzasadnienie: klasy mają funkcje wirtualne po to, aby można było ich używać za pośrednictwem interfejsu klasy bazowej. Funkcja znająca obiekt tylko poprzez klasę bazową może go usunąć, a klasa podrzędna musi mieć możliwość posprzątania (za pomocą swojego destruktora).

z503: konstruktor przyjmujący tylko jeden argument musi być zadeklarowany jako `explicit`.

Uzasadnienie: aby uniknąć zaskakujących niejawnych konwersji.

Systemy o ostrych ograniczeniach czasowych

Z800: Nie należy używać wyjątków.

Uzasadnienie: wyjątki są nieprzewidywalne.

Z801: operator `new` może być używany tylko przy rozruchu.

Uzasadnienie: operator `new` jest nieprzewidywalny.

Wyjątek: operator `new` wstawiający (o standardowym znaczeniu) może być używany na rzecz pamięci na stosie.

Z802: nie należy używać operatora `delete`.

Uzasadnienie: operator `delete` jest nieprzewidywalny i może doprowadzić do fragmentacji pamięci.

Z803: nie należy stosować operacji `dynamic_cast`.

Uzasadnienie: operacja `dynamic_cast` jest nieprzewidywalna (przyjmując typową technikę implementacji).

Z804: nie należy używać kontenerów biblioteki standardowej, z wyjątkiem `std::array`.

Uzasadnienie: są nieprzewidywalne (przyjmując typową technikę implementacji).

Systemy o krytycznym znaczeniu

Z900: operacje inkrementacji i dekrementacji nie mogą być używane jako podwyrażenia.

Przykład:

```
int x = v[++i]; // źle
```

Przykład:

```
++i;
int x = v[i]; // dobrze
```

Uzasadnienie: łatwo przeoczyć taką inkrementację.

Z901: kod nie powinien być uzależniony od reguły kolejności wykonywania działań poza działaniami arytmetycznymi.

Przykład:

```
x = a*b+c; // dobrze
```

Przykład:

```
if ( a<b || c<=d ) // Źle: (a<b) i (c<=d) powinny być w nawiasach
```

Uzasadnienie: programiści z małym doświadczeniem w językach C i C++ często mylą się w tych kwestiach.

Pozostawiliśmy luki w numeracji, aby móc w przyszłości dodać nowe zasady bez zmieniania istniejących numerów. Ludzie bardzo często zapamiętują zasady wg numerów, przez co zmiana numeracji mogłaby wywołać sprzeciwy.

25.6.3. Prawdziwe standardy kodowania

Istnieje wiele standardów pisania kodu w języku C++. Większość z nich to dokumenty wewnętrzne firm, które nie są szeroko dostępne. W wielu przypadkach są to wartościowe rzeczy, ale pewnie nie dla programistów tych firm. Poniżej znajduje się lista standardów, które — jeśli zostaną zastosowane w dziedzinach, dla których je stworzono — mogą się przydać:

Google C++ Style Guide (<https://google.github.io/styleguide/cppguide.html>). Raczej konserwatywny i restrykcyjny przewodnik stylu, który cały czas jest poprawiany.

Lockheed Martin Corporation, *Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program*, Document Number 2RDU00001 Rev C, grudzień 2005. Potoczna nazwa to *JSF++*. Zbiór zasad napisanych w firmie Lockheed-Martin Aero dla programistów urządzeń samolotowych. Zasady te zostały napisane przez prawdziwych programistów dla programistów piszących oprogramowanie urządzeń, od których zależy ludzkie życie — www.stroustrup.com/JSF-AV-rules.pdf.

Programming Research. Podręcznik pisania wysoce zintegrowanego kodu w języku C++, wersja 2.4; www.programmingresearch.com.

Sutter Herb, Alexandrescu Andrei, *Język C++. Standardy kodowania. 101 zasad, wytycznych i zalecanych praktyk*, Helion, 2005. Jest to bardziej coś w rodzaju „metastandardu kodowania”, tzn. zamiast konkretnych zasad zawiera ogólne wskazówki na temat tego, które zasady są dobre i dlaczego.

Należy zaznaczyć, że nie da się zastąpić znajomości dziedziny, języka programowania i odpowiednich technik programistycznych. W większości przypadków — zwłaszcza jeśli chodzi o programowanie systemów wbudowanych — trzeba także znać system operacyjny i architekturę sprzętu. Jeśli masz zamiar wykorzystać język C++ do programowania niskopoziomowego, przeczytaj raport komisji standaryzacyjnej tego języka na temat wydajności (ISO/IEC TR 18015, www.stroustrup.com/performanceTR.pdf). Pod słowem „wydajność” rozumiemy tu „programowanie systemów wbudowanych”.

W świecie systemów wbudowanych królują dialekty różnych języków programowania i języki specjalistyczne, ale gdzie tylko się da, stosuj standardowe narzędzia, biblioteki i języki (np. C++ i ISO). W ten sposób skrócisz czas nauki i zwiększysz szanse, że Twoja praca przetrwa dłużej.

Ćwiczenia

1. Uruchom poniższy program:

```
int v = 1; for (int i = 0; i<sizeof(v)*8; ++i) { cout << v << ' '; v <<=1; }
```
2. Uruchom powyższy program jeszcze raz, tylko `v` zadeklaruj jako `unsigned int`.
3. Za pomocą literałów szesnastkowych zdefiniuj następujące liczby typu `short unsigned int`:
 - a. z ustawionym każdym bitem,
 - b. z ustawionym najniższym (najmniej znaczącym) bitem,
 - c. z ustawionym najwyższym (najbardziej znaczącym) bitem,
 - d. z ustawionym całym najbardziej znaczącym bajtem,
 - e. z ustawionym całym najmniej znaczącym bajtem,
 - f. z ustawionym co drugim bitem (i najniższy bit powinien być 1),
 - g. z ustawionym co drugim bitem (i najniższy bit powinien być 0).
4. Wydrukuj każdy w formacie dziesiętnym i szesnastkowym.
5. Wykonaj ćwiczenia 3. i 4. przy użyciu operatorów bitowych (`|`, `&`, `<<`) oraz tylko literałów 1 i 0.

Powtórzenie

1. Co to jest system wbudowany? Podaj dziesięć przykładów, z których co najmniej trzy nie zostały wymienione w tej książce.
2. Czym wyróżniają się systemy wbudowane? Wymień pięć rzeczy, o których należy pamiętać.
3. Zdefiniuj przewidywalność w kontekście systemów wbudowanych.
4. Czemu utrzymanie i naprawa systemu wbudowanego mogą być trudne?
5. Czemu optymalizacja wydajności systemu może być złym pomysłem?
6. Dlaczego wolimy kod na wyższym poziomie abstrakcji niż niskopoziomowy?
7. Co to są błędy przejściowe? Czemu bardzo się ich boimy?
8. Jak można zaprojektować system, aby przetrwał awarię?
9. Czemu nie da się zapobiec wszystkim awariom?
10. Co to jest wiedza z danej dziedziny? Podaj przykłady dziedzin zastosowań.
11. Czemu potrzebna jest wiedza z danej dziedziny, aby zaprogramować system wbudowany?
12. Co to jest podsystem? Podaj kilka przykładów.
13. Jakie trzy rodzaje pamięci wyróżnia się w języku C++?
14. Kiedy należy używać pamięci wolnej?
15. Czemu w wielu systemach wbudowanych nie da się korzystać z pamięci wolnej?
16. Kiedy można bezpiecznie użyć operatora `new` w systemie wbudowanym?
17. Jakie potencjalne problemy może spowodować typ `std::vector` w systemach wbudowanych?
18. Jakie potencjalne problemy może spowodować użycie wyjątków w systemach wbudowanych?
19. Co to jest rekurencyjne wywołanie funkcji? Dlaczego niektórzy programiści systemów wbudowanych ich unikają? Czego używają w zamian?

20. Co to jest fragmentacja pamięci?
21. Co to jest system zbierania nieużytków (ang. *garbage collector*)?
22. Co to jest wyciek pamięci? Jakie problemy może powodować?
23. Co to jest zasób? Podaj kilka przykładów.
24. Co to jest wyciek zasobów i jak można mu systematycznie zapobiegać?
25. Dlaczego nie da się łatwo przenosić obiektów z jednego miejsca w pamięci do innego?
26. Co to jest stos?
27. Co to jest pula?
28. Czemu korzystanie ze stosów i pul powoduje fragmentację pamięci?
29. Dlaczego operacja `reinterpret_cast` jest niezbędna? Czemu jest wredna?
30. Czemu wskaźniki stosowane jako argumenty funkcji są niebezpieczne? Podaj kilka przykładów.
31. Jakie problemy może spowodować używanie wskaźników i tablic? Podaj kilka przykładów.
32. Czego można używać zamiast wskaźników (na tablice) w interfejsach?
33. Jak brzmi „pierwsze prawo informatyki”?
34. Co to jest bit?
35. Co to jest bajt?
36. Ile zwykle bitów mieści bajt?
37. Jakie działania można wykonywać na zbiorach bitów?
38. Co to jest „lub wykluczające” i do czego służy?
39. Jak można reprezentować zbiór (sekwencję lub cokolwiek) bitów?
40. Jaka jest konwencjonalna numeracja bitów w słowie?
41. Jaka jest konwencjonalna numeracja bajtów w słowie?
42. Co to jest słowo?
43. Ile zwykle bitów mieści się w słowie?
44. Co jest dziesiętnym odpowiednikiem wartości `0xf7`?
45. Jaką sekwencję bitów reprezentuje wartość `0xab`?
46. Co to jest typ `bitset` i do czego służy?
47. Czym różni się typ `unsigned int` od `signed int`?
48. W jakich sytuacjach preferowany jest typ `unsigned int` zamiast `signed int`?
49. Jak napisać pętlę, która musi przemierzyć bardzo dużą liczbę elementów?
50. Jaka będzie wartość liczby typu `unsigned int` po przypisaniu do niej wartości `-3`?
51. Kiedy stosuje się operacje bitowe i bajtowe (zamiast używać typów wyższego poziomu)?
52. Co to jest pole bitowe?
53. Do czego służą pola bitowe?
54. Na czym polega szyfrowanie? Po co się to robi?
55. Czy można zaszyfrować zdjęcie?
56. Co oznacza akronim TEA?

57. Jak wysyła się na wyjście liczby w formacie szesnastkowym?
58. Do czego służą standardy kodowania? Wymień kilka powodów do ich przestrzegania.
59. Czemu nie można utworzyć uniwersalnego standardu kodowania?
60. Wymień kilka cech dobrego standardu kodowania.
61. Jakie szkody może spowodować standard kodowania?
62. Sporządź listę przynajmniej dziesięciu zasad kodowania, które lubisz (uważasz za przydatne). Czemu są wg Ciebie przydatne?
63. Dlaczego należy unikać pisania nazw WIELKIMI_LITERAMI?

Terminologia

adres	ostre ograniczenia czasowe	system wbudowany
bit	pole bitowe	szyfrowanie
bitset	przewidywalność	unsigned
czas rzeczywisty	pula	wyciek
gadżet	standard kodowania	wykluczające lub
łagodne ograniczenia czasowe	system usuwania nieużytków	zasób

Praca domowa

1. Jeśli jeszcze tego nie zrobiłeś, rozwiąż wszystkie zadania Wypróbuj.
2. Zainicjuj 32-bitowe liczby całkowite ze znakiem tymi wzorcami bitowymi i wydrukuj wynik: wszystkie zera, wszystkie jedyńki, naprzemiennie zera i jedyńki (zaczynając od pierwszej jedyńki po lewej stronie), naprzemiennie jedyńki i zera (zaczynając od pierwszego zera po lewej stronie), wzorec 110011001100..., wzorec 001100110011..., wzorec zawierający bajty samych jedynek i samych zer zaczynający się od jedynek, wzorec zawierający bajty samych jedynek i samych zer zaczynający się od zer. Powtórz to ćwiczenie na 32-bitowych liczbach całkowitych bez znaku.
3. Dodaj bitowe operatory logiczne &, |, ^ oraz ~ do kalkulatora z rozdziału 7.
4. Napisz pętlę nieskończoną. Uruchom ją.
5. Napisz taką pętlę nieskończoną, którą trudno rozpoznać, że jest nieskończona. Może to być pętla, która kończy działanie w wyniku całkowitego zużycia pamięci.
6. Wydrukuj na wyjściu wartości szesnastkowe od 0 do 400 oraz od -200 do 200.
7. Wydrukuj na wyjściu wartość liczbową wszystkich znaków z klawiatury.
8. Nie używając żadnych standardowych nagłówków (np. <limits>) ani dokumentacji, oblicz liczbę bitów w typie int oraz sprawdź, czy typ char ma znak czy nie.
9. Spójrz na przykład z polami bitowymi w punkcie 25.5.5. Napisz program inicjalizujący PPN, który następnie wczytuje i drukuje wartość każdego pola, później zmienia wartość każdego pola (poprzez przypisanie) oraz drukuje wynik. Powtórz to zadanie, zapisując informacje PPN w 32-bitowej liczbie całkowitej bez znaku i użyj operatorów bitowych (punkt 25.5.4) do uzyskania dostępu do bitów w słowie.
10. Powtórz powyższe zadanie, zapisując bity w strukturze `bitset<32>`.
11. Wydrukuj czysty tekst z przykładu w punkcie 25.5.6.

12. Wykorzystaj algorytm TEA (punkt 25.5.6) do bezpiecznego połączenia dwóch komputerów. Wysłanie wiadomości e-mail to minimum.
13. Zaimplementuj prosty wektor mogący przechowywać najwyżej N elementów alokowanych w puli. Przetestuj go dla $N=1000$ i elementów całkowitoliczbowych.
14. Zmierz czas (punkt 26.6.1) potrzebny do alokowania za pomocą operatora `new` 10 000 obiektów różnych rozmiarów w przedziale $[0,1000)$ bajtów. Następnie sprawdź, ile czasu zajmuje ich dealokacja za pomocą operatora `delete`. Zrób to dwa razy. Raz dokonaj dealokacji w kolejności odwrotnej do alokacji, a drugi raz w losowej kolejności. Następnie podobnie zmierz czas alokowania 10 000 obiektów o rozmiarze 500 bajtów w puli oraz czas ich dealokowania. Zrób to samo dla 10 000 obiektów o losowych rozmiarach w przedziale $[0,1000)$ bajtów alokowanych na stosie i dealokowanych w kolejności odwrotnej do alokacji. Porównaj wyniki. Wykonaj pomiary przynajmniej trzy razy, aby upewnić się, że wyniki są wiarygodne.
15. Sformułuj 20 zasad stylu pisania kodu (nie ściągać z podrozdziału 25.6). Zastosuj je w jakimś napisanym przez siebie programie składającym się z przynajmniej 300 wierszy kodu. Napisz krótki (1 do 2 stron) komentarz na temat wrażeń wyniesionych ze stosowania tych zasad w swoim programie. Znalazłeś jakieś błędy w kodzie? Czy kod stał się bardziej przejrzysty? Czy niektóre partie straciły klarowność? Wyciągając wnioski z własnych doświadczeń, zmodyfikuj swój zestaw zasad.
16. W punktach 25.4.3 i 25.4.4 opisaliśmy klasę `Array_ref`, która według naszych zapewnień ma dawać łatwiejszy i bezpieczniejszy dostęp do elementów tablicy. Zwłaszcza twierdziliśmy, że poprawnie obsługiwane będzie dziedziczenie. Spróbuj na kilka sposobów zapisać typ `Rectangle*` w kontenerze `vector<Circle*>`, używając `Array_ref<Shape*>`, ale bez żadnych rzutowań i innych operacji związanych z niezdefiniowanym zachowaniem. Powinno być niemożliwe.

Podsumowanie

Czy można powiedzieć, że programowanie systemów wbudowanych zasadniczo opiera się na „zabawach z bitami”? Zdecydowanie nie, zwłaszcza jeśli celowo próbuje się zminimalizować ilość operacji na bitach, traktując je jako potencjalne zagrożenie poprawności. Jednak operowanie na bitach jest konieczne gdzieś w systemie. Kwestia dotyczy tylko tego, gdzie i jak to robić. W większości systemów można, a nawet trzeba zlokalizować niskopoziomowy kod. Większość ciekawych systemów, z którymi mamy do czynienia, to systemy wbudowane, i większość najciekawszych i najtrudniejszych zadań programistycznych pochodzi z tej dziedziny.



