

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

C++. Zaawansowane programowanie

Autorzy: Nicholas A. Solter, Scott J. Kleper

Tłumaczenie: Paweł Gonera, Rafał Jońca, Tomasz Żmijewski

ISBN: 83-7361-998-4

Tytuł oryginału: [Professional C++](#)

Format: B5, stron: 912



Poznaj możliwości i mechanizmy C++ stosowane przez profesjonalistów

- Efektywne zarządzanie pamięcią
- Obsługa błędów
- Biblioteki i wzorce projektowe

Język C++ od dawna cieszy się zasłużoną popularnością wśród twórców oprogramowania. Jednak często nawet najbardziej doświadczeni programiści nie wykorzystują wszystkich jego możliwości. W opracowaniach dotyczących tego języka najczęściej omawiana jest składnia i elementy języka, a znacznie rzadziej – jego praktyczne zastosowania. Brak odniesień do praktyki powoduje nieznaną wielu niezwykle przydatnych i ułatwiających pracę aspektów C++ i nadmierne eksploatawanie prostych, książkowych konstrukcji.

Książka „C++. Zaawansowane programowanie” to przegląd metod programowania nie omawianych w większości publikacji. Ten przeznaczony dla średnio zaawansowanych programistów podręcznik przedstawia zasady stosowania C++ do rozwiązywania prawdziwych problemów. Opisuje nieznane możliwości znacznie przyspieszające i usprawniające pracę, sposoby tworzenia przenośnego kodu oraz testowania oprogramowania. Książka zawiera również omówienie biblioteki standardowej C++ oraz wzorców projektowych.

- Zasady tworzenia projektów w C++
- Projektowanie obiektowe
- Korzystanie z biblioteki standardowej C++
- Tworzenie kodu przeznaczonego do wielokrotnego zastosowania
- Styl kodowania
- Zarządzanie pamięcią
- Procedury obsługi wyjątków
- Przeciążanie operatorów
- Tworzenie aplikacji wieloplatformowych
- Testowanie kodu i usuwanie błędów
- Programowanie rozproszone
- Stosowanie wzorców projektowych

Nauč się korzystać z C++ tak, jak robią to profesjonalści



Spis treści

0 autorach	17
Wprowadzenie	19
Część I Wprowadzenie do profesjonalnego C++	23
Rozdział 1. Skrócony kurs C++	25
Podstawy C++	25
Teraz obowiązkowo Hello, World	26
Przestrzeń nazw	28
Zmienne	30
Operatory	32
Typy danych	34
Wyrażenia warunkowe	36
Pętle	38
Tablice	40
Funkcje	41
To był elementarz	42
C++ z bliska	42
Wskaźniki i pamięć alokowana dynamicznie	42
Łańcuchy znakowe w C++	45
Referencje	47
Wyjątki	48
Wiele zastosowań słowa kluczowego const	49
C++ jako język obiektowy	50
Deklarowanie klasy	51
Nasz pierwszy praktyczny program C++	53
System obsługi kadrowej	53
Klasa Employee	54
Klasa Database	58
Interfejs użytkownika	62
Udoskonalanie programu	65
Podsumowanie	65
Rozdział 2. Tworzenie profesjonalnych programów w C++	67
Czym jest projekt programu?	68
Dlaczego projekt programu jest tak ważny?	68
Co jest szczególnego w projektach C++?	70
Dwie zasady tworzenia projektów w C++	71
Abstrakcja	71
Wielokrotne użycie	73

Projektowanie programu szachowego	75
Wymagania	75
Kolejne etapy projektowania	76
Podsumowanie	80
Rozdział 3. Projektowanie obiektowe	83
Obiektowe spojrzenie na świat	83
Czy myślę proceduralnie?	83
Filozofia obiektowa	84
W świecie obiektów	87
Powiązania obiektów	89
Abstrakcja	99
Podsumowanie	102
Rozdział 4. Użycie bibliotek i wzorców	103
Wielokrotne użycie kodu	103
Uwaga na temat terminologii	104
Podejmowanie decyzji, czy ponownie wykorzystać kod	105
Strategie wielokrotnego użycia kodu	107
Korzystanie z obcych aplikacji	112
Biblioteki open-source	113
Standardowa biblioteka C++	114
Projektowanie z użyciem wzorców i technik	127
Techniki projektowe	128
Wzorce projektowe	129
Podsumowanie	130
Rozdział 5. Projektowanie z myślą o ponownym użyciu	131
Filozofia wielokrotnego użycia	132
Jak projektować kod wielokrotnego użycia	132
Użycie abstrakcji	133
Struktura kodu wielokrotnego użycia	134
Projektowanie interfejsów	139
Łączenie ogólności i prostoty użycia	145
Podsumowanie	146
Rozdział 6. Użycie metod inżynierii oprogramowania	147
Potrzeba stosowania procesu	147
Modele cyklu życia oprogramowania	149
Model etapowy i kaskadowy	149
Metoda spirali	152
Rational Unified Process	154
Metodyki inżynierii oprogramowania	155
Programowanie ekstremalne (XP)	156
Sąd nad oprogramowaniem	160
Tworzenie własnego procesu i metodyki	161
Bądź otwarty na nowe pomysły	161
Dziel się pomysłami	161
Zorientuj się, co działa, a co nie	162
Nie bądź renegatem	162
Podsumowanie	162

Część II Profesjonalne kodowanie w C++	163
Rozdział 7. Styl kodowania	165
Piękny wygląd	165
Myślenie o przyszłości	165
Zachowanie porządku	166
Elementy dobrego stylu	166
Dokumentowanie kodu	166
Po co pisać komentarze	166
Style komentowania	169
Komentarze w tej książce	174
Dekompozycja	174
Dekompozycja przez refaktoring	174
Dekompozycja na etapie projektu	176
Dekompozycja w tej książce	176
Nazewnictwo	176
Dobór właściwej nazwy	177
Konwencje nazewnictwa	177
Elementy języka a styl	180
Stałe	180
Korzystanie ze zmiennych const	180
Użycie referencji zamiast wskaźników	180
Wyjątki użytkownika	181
Formatowanie	181
Rozmieszczenie nawiasów klamrowych	182
Spacje i nawiasy	183
Spacje i tabulatory	183
Wyzwania związane ze stylem	183
Podsumowanie	184
Rozdział 8. Poznajemy lepiej klasy i obiekty	185
Wprowadzenie do przykładu arkusza kalkulacyjnego	185
Pisanie klas	186
Definicje klasy	186
Definiowanie metod	188
Użycie obiektów	192
Cykl życia obiektów	193
Tworzenie obiektu	193
Usuwanie obiektu	204
Przypisanie do obiektów	206
Odróżnianie kopiowania od przypisania	208
Podsumowanie	210
Rozdział 9. Doskonalenie znajomości klas i obiektów	211
Dynamiczna alokacja pamięci w obiektach	211
Klasa Spreadsheet	212
Zwalnianie pamięci w destruktorach	213
Obsługa kopiowania i przypisania	214
Różne rodzaje pól w klasach	221
Pola statyczne	221
Pola stałe	223

Pola będące referencjami	225
Pola będące stałymi referencjami	226
Jeszcze o metodach	226
Metody statyczne	226
Metody stałe	227
Przeciążanie metod	229
Parametry domyślne	230
Metody inline	231
Klasy zagnieżdżone	233
Zaprzyjaźnienie	235
Przeciążanie operatorów	236
Implementacja dodawania	236
Przeciążanie operatorów arytmetycznych	240
Przeciążanie operatorów porównania	242
Tworzenie typów z przeciążaniem operatorów	244
Wskaźniki pól i metod	244
Tworzenie klas abstrakcyjnych	245
Użycie klas interfejsu i implementacji	246
Podsumowanie	249
Rozdział 10. Przygoda z dziedziczeniem	251
Klasy i dziedziczenie	251
Rozszerzanie klas	252
Nadpisywanie metod	255
Dziedziczenie a wielokrotne użycie kodu	258
Klasa WeatherPrediction	258
Nowe możliwości w podklasie	259
Inne możliwości w podklasie	261
Szacunek dla rodziców	262
Konstruktory rodziców	262
Destruktry rodziców	264
Odwołania do danych rodziców	266
Rzutowanie w górę i w dół	267
Dziedziczenie i polimorfizm	268
Powrót do arkusza kalkulacyjnego	269
Polimorficzna komórka arkusza	269
Klasa bazowa komórki arkusza	270
Poszczególne podklasy	272
Polimorfizm dla zaawansowanych	274
Uwagi na przyszłość	275
Dziedziczenie wielokrotne	276
Dziedziczenie po wielu klasach	277
Kolizje nazw i niejednoznaczne klasy bazowe	278
Ciekawostki i tajemnice dziedziczenia	281
Zmiana charakterystyki nadpisanej metody	281
Przypadki szczególne nadpisywania metod	285
Konstruktory kopiujące i operator równości	291
Prawda o wirtualności	293
Typy podczas działania programu	296
Dziedziczenie niepubliczne	297
Wirtualne klasy bazowe	298
Podsumowanie	299

Rozdział 11. Szablony i kod ogólny	301
Szablony w skrócie	302
Szablony klas	303
Pisanie szablonu klasy	303
Jak kompilator przetwarza szablony	311
Rozdzielanie kodu szablonu między pliki	312
Parametry szablonów	313
Szablony metod	316
Specjalizacja klas szablonów	321
Podklasy szablonów klas	324
Dziedziczenie a specjalizacja	325
Szablony funkcji	326
Specjalizacja szablonów funkcji	327
Przeciążanie szablonów funkcji	327
Szablony funkcji zaprzyjaźnionych szablonów klas	328
Szablony: zagadnienia zaawansowane	330
Więcej o parametrach szablonów	330
Częściowa specjalizacja szablonów klas	338
Przeciążanie jako symulacja specjalizacji częściowej funkcji	344
Szablony rekurencyjnie	345
Podsumowanie	353
Rozdział 12. Dziwactwa C++	355
Referencje	355
Zmienne referencyjne	356
Referencje do pól	358
Parametry referencyjne	358
Zwracanie wartości przez referencję	359
Wybór między referencjami a wskaźnikami	360
Zamieszanie ze słowami kluczowymi	362
Słowo kluczowe const	362
Słowo kluczowe static	365
Porządek inicjalizacji zmiennych nielokalnych	369
Typy i rzutowanie	369
typedef	370
Rzutowanie	371
Zasięg	375
Pliki nagłówkowe	376
Narzędzia C	377
Listy parametrów o zmiennej długości	378
Makra preprocesora	380
Podsumowanie	381
Część III Zaawansowane elementy C++	383
Rozdział 13. Skuteczne zarządzanie pamięcią	385
Użycie pamięci dynamicznej	385
Jak sobie wyobrazić pamięć	386
Alokacja i zwalnianie	387
Tablice	389
Użycie wskaźników	396

Dualność tablic i wskaźników	398
Tablice są wskaźnikami!	398
Nie wszystkie wskaźniki są tablicami!	399
Łańcuchy dynamiczne	400
Łańcuchy w stylu C	400
Literały łańcuchowe	402
Klasa C++ string	402
Niskopoziomowe operacje na pamięci	404
Arytmetyka wskaźników	405
Specjalne zarządzanie pamięcią	405
Odśmiecanie pamięci	406
Pule obiektów	407
Wskaźniki funkcji	407
Typowe błędy związane z zarządzaniem pamięcią	409
Brak pamięci na łańcuchy	409
Wycieki pamięci	410
Dwukrotne usuwanie i błędne wskaźniki	413
Dostęp do pamięci spoza zakresu	414
Podsumowanie	414
Rozdział 14. Wejście-wyjście w C++. Pożegnanie z mitami	415
Użycie strumieni	416
Czym jest strumień	416
Źródła i przeznaczenie strumieni	416
Strumienie wyjściowe	417
Strumienie wejściowe	421
Wejście i wyjście realizowane za pomocą obiektów	426
Strumienie łańcuchowe	428
Strumienie plikowe	429
Poruszanie się za pomocą seek() i tell()	430
Wiązanie strumieni	432
Wejście-wyjście dwukierunkowe	433
Internacjonalizacja	435
Znaki rozszerzone	435
Zestawy znaków inne niż zachodnie	436
Ustawienia lokalne i fazy	436
Podsumowanie	438
Rozdział 15. Obsługa błędów	439
Błędy i wyjątki	440
Czym są wyjątki	440
Zalety wyjątków języka C++	441
Wady wyjątków języka C++	442
Zalecenia	442
Mechanika wyjątków	443
Zgłaszanie i wyłapywanie wyjątków	443
Typy wyjątków	445
Zgłaszanie i wyłapywanie wielu wyjątków	447
Niewyłapane wyjątki	449
Lista zgłaszanych wyjątków	451

Wyjątki i polimorfizm	455
Hierarchia standardowych wyjątków	455
Przechwytywanie wyjątków w hierarchii klas	456
Pisanie własnych klas wyjątków	458
Odwijanie stosu i zwalnianie zasobów	461
Złap wyjątek, dokonaj zwolnienia zasobu i ponownie zgłoś wyjątek	462
Wykorzystanie inteligentnych wskaźników	463
Typowe zagadnienia dotyczące obsługi błędów	463
Błędy alokacji pamięci	463
Błędy w konstruktorach	466
Błędy w destruktorze	467
Połączenie wszystkiego razem	468
Podsumowanie	470

Część IV Pozbywanie się błędów471

Rozdział 16. Przeciążanie operatorów języka C++ 473

Omówienie przeciążania operatorów	474
Powody przeciążania operatorów	474
Ograniczenia przeciążania operatorów	474
Wybór przeciążania operatorów	475
Operatory, których nie należy przeciążać	478
Podsumowanie operatorów z możliwością przeciążania	478
Przeciążanie operatorów arytmetycznych	481
Przeciążanie jednoargumentowego operatora plus i minus	481
Przeciążanie inkrementacji i dekrementacji	482
Przeciążanie operatorów bitowych i operatorów logicznych dwuargumentowych	484
Przeciążanie operatorów wstawiania i wydobywania	484
Przeciążanie operatora indeksu tablicy	486
Zapewnienie dostępu tylko do odczytu dla operatora operator[]	489
Indeksy niebędące liczbami całkowitymi	490
Przeciążenie operatora wywołania funkcji	491
Przeciążanie operatorów dereferencji	493
Implementacja operatora operator*	494
Implementacja operatora operator->	495
Co oznacza operator->*?	496
Pisanie operatorów konwersji	496
Problemy niejednoznaczności operatorów konwersji	498
Konwersje dla wyrażeń logicznych	498
Przeciążanie operatorów alokacji i zwalniania pamięci	500
Jak tak naprawdę działają operatory new i delete	501
Przeciążanie operatorów new i delete	502
Przeciążanie operatorów new i delete z dodatkowymi parametrami	505
Podsumowanie	507

Rozdział 17. Pisanie wydajnego kodu 509

Omówienie wydajności i efektywności	509
Dwa podejścia do efektywności	510
Dwa rodzaje programów	510
Czy C++ to nieefektywny język programowania?	510

Efektywność na poziomie języka	511
Wydajna obsługa obiektów	512
Nie nadużywaj kosztownych elementów języka	515
Użycie metod i funkcji rozwijanych w miejscu wywołania	516
Efektywność na poziomie projektu	517
Stosuj buforowanie podręczne, jeśli tylko to możliwe	517
Pule obiektów	518
Wykorzystanie puli wątków	523
Profilowanie	524
Profilowanie przykładu za pomocą gprof	524
Podsumowanie	533
Rozdział 18. Tworzenie aplikacji wieloplatformowych i wielojęzycznych	535
Tworzenie aplikacji wieloplatformowych	536
Kwestie architektury	536
Kwestie implementacji	539
Elementy specyficzne dla platformy programowo-sprzętowej	540
Aplikacje wielojęzyczne	541
Mieszanie języków C i C++	541
Zmieszane paradygmaty	542
Konsolidacja kodu języka C	545
Łączenie Javy i C++ dzięki JNI	546
Mieszanie C++ z Perlem i skryptami powłoki	549
Mieszanie C++ z kodem w języku asemblera	552
Podsumowanie	552
Rozdział 19. Podstawy testowania	553
Kontrola jakości	554
Kto jest odpowiedzialny za testowanie?	554
Cykl życia błędu	554
Narzędzia do śledzenia błędów	555
Testy jednostkowe	556
Metody testowania jednostkowego	557
Proces testowania jednostkowego	558
Testy jednostkowe w praktyce	562
Testowanie wyższego poziomu	570
Testy integracyjne	571
Testy systemowe	572
Testy regresyjne	573
Wskazówki na temat testowania	574
Podsumowanie	574
Rozdział 20. Wszystko o debugowaniu	575
Podstawowe zasady debugowania	575
Taksonomia błędów	576
Unikanie błędów	576
Przewidywanie błędów	577
Rejestrowanie błędów	577
Rejestrowanie śladu	578
Asercje	589

Techniki debugowania	590
Reprodukcja błędów	591
Debugowanie błędów powtarzalnych	592
Debugowanie błędów niemożliwych do powtórzenia	592
Debugowanie błędów pamięci	593
Debugowanie programów wielowątkowych	597
Przykład debugowania: cytowanie artykułów	598
Wnioski z przykładu ArticleCitations	609
Podsumowanie	609
Rozdział 21. W głąb STL: kontenery i iteratory	611
Przegląd kontenerów	612
Wymagania co do elementów	612
Wyjątki i kontrola błędów	614
Iteratory	614
Kontenery sekwencyjne	616
Wektor	616
Specjalizowana implementacja <code>vector<bool></code>	634
Kontener deque	635
Kontener list	636
Adaptory kontenerów	640
Adapter queue	640
Adapter priority_queue	643
Adapter stack	646
Kontenery asocjacyjne	647
Klasa narzędziowa pair	647
Kontener map	648
Kontener multimap	657
Kontener set	661
Kontener multiset	663
Inne kontenery	663
Tablice jako kontenery STL	663
string jako kontener STL	664
Strumienie jako kontenery STL	665
Kontener bitset	665
Podsumowanie	670
Część V Użycie bibliotek i wzorców	671
Rozdział 22. Poznajemy algorytmy STL oraz obiekty funkcyjne	673
Przegląd algorytmów	674
Algorytmy <code>find()</code> oraz <code>find_if()</code>	674
Algorytm <code>accumulate()</code>	677
Obiekty funkcyjne	678
Obiekty funkcji arytmetycznych	678
Obiekty funkcji porównań	679
Obiekty funkcji logicznych	680
Adaptory obiektów funkcyjnych	681
Tworzenie własnych obiektów funkcyjnych	685

Szczegóły budowy algorytmów	686
Algorytmy użytkowe	686
Algorytmy niemodyfikujące	687
Algorytmy modyfikujące	693
Algorytmy sortujące	698
Algorytmy dla zbiorów	701
Przykład użycia algorytmów i obiektów funkcyjnych: kontrola list wyborców	702
Problem kontroli rejestrów wyborców	703
Funkcja auditVoterRolls()	703
Funkcja getDuplicates()	704
Funktor RemoveNames	705
Funktor NameInList	706
Testowanie funkcji auditVoterRolls()	707
Podsumowanie	708

Rozdział 23. Dostosowywanie i rozszerzanie STL **709**

Alokatory	710
Adaptory iteratorów	710
Iteratory działające wstecz	710
Iteratory strumienia	712
Iteratory wstawiające	712
Rozszerzanie STL	714
Po co rozszerzamy STL?	715
Tworzenie algorytmów STL	715
Tworzenie kontenera STL	717
Podsumowanie	747

Rozdział 24. Rzecz o obiektach rozproszonych **749**

Zalety programowania rozproszonego	749
Rozpraszenie w celu zwiększenia skalowalności	749
Rozpraszenie w celu zwiększenia niezawodności	750
Rozpraszenie w celu centralizacji	750
Rozpraszenie treści	751
Przetwarzanie rozproszone a sieciowe	751
Obiekty rozproszone	752
Serializacja i szeregowanie	752
Zdalne wywoływanie procedur	756
CORBA	758
Język definicji interfejsu	758
Implementacja klasy	761
Wykorzystanie obiektów	762
XML	766
Szybki kurs XML	766
XML jako technologia obiektów rozproszonych	768
Generowanie i analizowanie XML w C++	769
Kontrola poprawności XML	777
Tworzenie obiektów rozproszonych z użyciem XML	779
SOAP (ang. Simple Object Access Protocol)	782
Podsumowanie	784

Rozdział 25. Korzystanie z technik i bibliotek	785
Nigdy nie pamiętam, jak...	786
...utworzyć klasę	786
...dziedziczyć po istniejącej klasie	787
...zgłosić i przechwycić wyjątek	788
...odczytać z pliku	789
...zapisać do pliku	789
...utworzyć szablon klasy	790
Musí istnieć lepszy sposób	792
Inteligentne wskaźniki ze zliczaniem referencji	792
Podwójne rozsyłanie	797
Klasy domieszkowe	803
Biblioteki obiektowe	806
Wykorzystywanie bibliotek	806
Paradygmat model-widok-kontroler	807
Podsumowanie	808
Rozdział 26. Wykorzystanie wzorców projektowych	809
Wzorzec singleton	810
Przykład: mechanizm dziennika	810
Implementacja klasy singleton	810
Wykorzystanie klasy singleton	815
Wzorzec factory	816
Przykład: symulacja fabryki samochodów	816
Implementacja klasy factory	818
Wykorzystanie wzorca factory	820
Inne zastosowania wzorca factory	822
Wzorzec proxy	822
Przykład: ukrywanie problemów z połączeniem sieciowym	822
Implementacja klasy proxy	823
Wykorzystanie klasy proxy	824
Wzorzec adapter	824
Przykład: adaptacja biblioteki XML	824
Implementacja adaptera	825
Wykorzystanie klasy adaptera	828
Wzorzec decorator	829
Przykład: definiowanie stylów na stronach WWW	829
Implementacja klasy decorator	830
Wykorzystanie klasy decorator	831
Wzorzec chain of responsibility	832
Przykład: obsługa zdarzeń	833
Implementacja łańcucha odpowiedzialności	833
Wykorzystanie łańcucha odpowiedzialności	834
Wzorzec observer	834
Przykład: obsługa zdarzeń	835
Implementacja wzorca observer	835
Wykorzystanie wzorca observer	836
Podsumowanie	837

Dodatki 839**Dodatek A Rozmowy kwalifikacyjne z C++ 841**

Rozdział 1. „Skrócony kurs C++”	841
Rozdział 2. „Tworzenie profesjonalnych programów w C++”	842
Rozdział 3. „Projektowanie obiektowe”	843
Rozdział 4. „Użycie bibliotek i wzorców”	844
Rozdział 5. „Projektowanie z myślą o ponownym użyciu”	845
Rozdział 6. „Użycie metod inżynierii oprogramowania”	846
Rozdział 7. „Styl kodowania”	847
Rozdziały 8. i 9. Klasy i obiekty	848
Rozdział 10. „Przygoda z dziedziczeniem”	851
Rozdział 11. „Szablony i kod ogólny”	851
Rozdział 12. „Dziwactwa C++”	852
Rozdział 13. „Skuteczne zarządzanie pamięcią”	853
Rozdział 14. „Wejście-wyjście w C++. Pożegnanie z mitami”	854
Rozdział 15. „Obsługa błędów”	855
Rozdział 16. „Przeciążanie operatorów języka C++”	856
Rozdział 17. „Pisanie wydajnego kodu”	856
Rozdział 18. „Tworzenie aplikacji wieloplatformowych i wielojęzycznych”	857
Rozdział 19. „Podstawy testowania”	858
Rozdział 20. „Wszystko o debugowaniu”	859
Rozdziały 21., 22. i 23. Biblioteka STL	859
Rozdział 24. „Rzecz o obiektach rozproszonych”	860
Rozdział 25. „Korzystanie z technik i bibliotek”	861
Rozdział 26. „Wykorzystanie wzorców projektowych”	861

Dodatek B Bibliografia z omówieniami 863

C++	863
C++ dla początkujących	863
Różne informacje o C++	864
Strumienie wejścia-wyjścia	865
Standardowa biblioteka C++	866
Szablony C++	866
Język C	866
Integracja C++ z innymi językami	867
Algorytmy i struktury danych	867
Oprogramowanie open-source	868
Metodologie inżynierii programowania	868
Styl programowania	869
Architektura komputerów	869
Wydajność	870
Testowanie	870
Debugowanie	870
Obiekty rozproszone	870
CORBA	871
XML oraz SOAP	871
Wzorce projektowe	872

Skorowidz 873

8

Poznajemy lepiej klasy i obiekty

Jako język obiektowy C++ oferuje narzędzia do obsługi *obiektów* i pisania definicji obiektów, czyli *klas*. Można oczywiście napisać w C++ program bez klas i bez obiektów, ale w ten sposób sami rezygnujemy z najważniejszych możliwości języka. Pisanie programu w C++ bez klas to jak stołowanie się w McDonald'sie w trakcie podróży do Paryża! Aby dobrze wykorzystać klasy i obiekty, trzeba zrozumieć związaną z nimi składnię i poznać ich możliwości.

W rozdziale 1. powiedzieliśmy nieco o podstawowych elementach składni C++. W rozdziale 3. powiedzieliśmy o obiektowym programowaniu w C++ oraz pokazaliśmy strategię projektowania klas i obiektów. Teraz opiszemy najważniejsze pojęcia związane z klasami i obiektami: pisanie definicji klas, definiowanie metod, użycie obiektów na stosie i stercie, pisanie konstruktorów, konstruktory domyślne, konstruktory generowane przez kompilator, listy inicjalizacyjne w konstruktorach, konstruktory kopiujące, destruktory i operatory przypisania. Nawet osoby mające doświadczenie w pracy z klasami i obiektami powinny przynajmniej przejrzeć ten rozdział, gdyż mogą znaleźć w nim pewne specyficzne informacje, które mogą być dla nich nowością.

Wprowadzenie do przykładu arkusza kalkulacyjnego

W tym i następnym rozdziale omawiać będziemy działający przykład prostej aplikacji arkusza kalkulacyjnego. Arkusz taki to dwuwymiarowa siatka komórek, z których każda zawiera liczbę lub tekst. Profesjonalne arkusze kalkulacyjne, jak Microsoft Excel, umożliwiają robienie obliczeń matematycznych, na przykład wyliczanie sumy wartości ze zbioru komórek. Arkusz omawiany w tym rozdziale nie ma być konkurencją dla Excela, ale świetnie nadaje się do pokazania pewnych aspektów klas i obiektów.

Aplikacja arkusza ma dwie podstawowe klasy: Spreadsheet i SpreadsheetCell (odpowiednio arkusz i komórka). Każdy obiekt Spreadsheet zawiera obiekty SpreadsheetCell. Poza tym klasa SpreadsheetApplication zarządza różnymi obiektami Spreadsheet. W tym rozdziale skoncentrujemy się na komórkach, czyli klasie SpreadsheetCell, natomiast klasami Spreadsheet i SpreadsheetApplication zajmiemy się w rozdziale 9.

W rozdziale tym pokażemy kilka różnych wersji klasy `SpreadsheetCell`, aby nowe pojęcia omawiać stopniowo. Dlatego kolejne wersje klasy nie muszą wcale pokazywać najlepszych możliwych rozwiązań. Szczególnie w początkowych przykładach pomijamy ważne rzeczy, które w rzeczywistości powinny zostać zakodowane, ale jeszcze ich nie omówiliśmy. Ostateczną wersję klasy można pobrać tak, jak to opisano we wstępie do książki.

Pisanie klas

Kiedy piszemy klasę, opisujemy jej zachowania, czyli *metody*, które będą dostępne w obiektach tej klasy, oraz właściwości, czyli *pola danych*, które będzie zawierał każdy obiekt.

Klasę pisze się w dwóch etapach: najpierw definiuje się samą klasę, potem jej metody.

Definicje klasy

Oto pierwsza próba stworzenia prostej klasy `SpreadsheetCell`, gdzie każdej komórce można przypisać tylko jedną wartość:

```
// SpreadsheetCell.h
class SpreadsheetCell
{
public:
    void setValue(double inValue);
    double getValue();

protected:
    double mValue;
};
```

Jak mówiliśmy już w rozdziale 1., definicja klasy zaczyna się od słowa kluczowego `class` oraz nazwy klasy. Definicja klasy to *instrukcja* języka C++, więc musi kończyć się średnikiem. Jeśli tego średnika nie dodamy, kompilator prawdopodobnie zgłosi kilka błędów, z których większość będzie dotyczyła nie wiadomo czego.

Definicję klasy zwykle umieszcza się w pliku *NazwaKlasy.h*.

Metody i pola

Dwa wiersze wyglądające jak deklaracje funkcji to tak naprawdę deklaracje metod klasy:

```
void setValue(double inValue);
double getValue();
```

Wiersz wyglądający jak deklaracja zmiennej to deklaracja pola danych klasy:

```
double mValue;
```

Każdy obiekt będzie miał własną zmienną `mValue`. Jednak implementacja metod jest wspólna dla wszystkich obiektów. Klasy mogą zawierać dowolną liczbę metod i pól. Nie można polu nadać takiej samej nazwy, jak metodzie.

Kontrola dostępu

Każda metoda i każde pole klasy mieści się w zakresie jednego z trzech *określników dostępu*: `public`, `protected` lub `private`. Określniki te dotyczą wszystkich metod i pól znajdujących się za nimi, aż do następnego określnika. W przypadku naszej klasy metody `setValue()` i `getValue()` są publiczne (`public`), a pole `mValue` jest chronione (`protected`):

```
public:
    void setValue(double inValue);
    double getValue();
protected:
    double mValue;
```

Domyślnym określnikiem dostępu dla klasy jest `private`: wszystkie metody i pola znajdujące się przed pierwszym określnikiem dostępu są prywatne. Gdybyśmy na przykład przeimieśli określnik `public` pod metodę `setValue()`, metoda `setValue()` przestałaby być publiczna, a byłaby prywatna:

```
// SpreadsheetCell.h
class SpreadsheetCell
{
    void setValue(double inValue);
public:
    double getValue();

protected:
    double mValue;
};
```

W języku C++ struktury mogą mieć metody analogicznie jak klasy. Zresztą tak naprawdę jedyną różnicą między strukturami a klasami polega na tym, że w strukturach domyślnym określnikiem dostępu jest `public`, a nie `private`, jak w klasach.

W poniższej tabeli zestawiono znaczenie wszystkich trzech określników dostępu.

Określnik dostępu	Znaczenie	Kiedy używać
<code>public</code>	Metody publiczne można wywołać z dowolnego kodu, tak samo jak sięgnąć do publicznego pola obiektu.	Do zachowań (metod) używanych przez klientów oraz do akcesorów (metod dostępu) do pól prywatnych i chronionych.
<code>protected</code>	Metody chronione są dostępne dla wszystkich metod danej klasy, tak samo wszystkie metody tej klasy mogą korzystać z pól chronionych. Metody <i>podklasy</i> (omówione w rozdziale 10.) także mogą wywoływać metody chronione nadklasy i sięgnąć do pól chronionych.	Metody pomocnicze, które nie powinny być dostępne dla klientów, oraz większość pól danych.

Określnik dostępu	Znaczenie	Kiedy używać
private	Jedynie metody tej samej klasy mają dostęp do metod prywatnych i pól prywatnych. Metody podklasy <i>nie</i> mogą korzystać z prywatnych metod i pól.	Jedynie wtedy, gdy chcemy ograniczyć dostęp także dla podklas.

Określniki dostępu są definiowane na poziomie klasy, a nie obiektu, więc metody klasy mogą sięgać do metod i pól chronionych dowolnego obiektu danej klasy.

Kolejność deklaracji

Metody, pola i określniki dostępu można deklarować w dowolnej kolejności: język C++ nie nakłada tu żadnych ograniczeń typu „metody muszą być przed polami, a sekcja public przed private. Co więcej, określniki dostępu mogą się powtarzać. Na przykład definicję klasy SpreadsheetCell można zapisać też tak:

```
// SpreadsheetCell.h
class SpreadsheetCell
{
public:
    void setValue(double inValue);

protected:
    double mValue;

public:
    double getValue();
};
```

Jednak aby zwiększyć czytelność kodu, dobrze jest grupować razem deklaracje public, protected i private, a w ich ramach grupować metody i pola. W niniejszej książce stosowana jest następująca kolejność definicji:

```
class NazwaKlasy
{
public:
    // deklaracje metod
    // deklaracje pól
protected:
    // deklaracje metod
    // deklaracje pól
private:
    // deklaracje metod
    // deklaracje pól
};
```

Definiowanie metod

Podana wcześniej definicja klasy SpreadsheetCell wystarczy, aby można było tworzyć obiekty tej klasy. Jeśli jednak spróbujemy wywołać metodę setValue() lub getValue(), konsolidator zgłosi błąd o braku definicji tych metod. Wynika to stąd, że w klasie podano prototypy

tych metod, ale nie ma nigdzie ich implementacji. Tak jak dla zwykłych funkcji pisze się prototyp i definicję, tak samo trzeba napisać prototyp i definicję metody. Zauważmy, że przed definicją metody musi pojawiać się nazwa klasy. Zwykle definicja klasy znajduje się w pliku nagłówkowym, a definicje metod są w pliku źródłowym zawierającym plik nagłówkowy. Oto definicje dwóch metod klasy `SpreadsheetCell`:

```
// SpreadsheetCell.cpp
#include "SpreadsheetCell.h"

void SpreadsheetCell::setValue(double inValue)
{
    mValue = inValue;
}

double SpreadsheetCell::getValue()
{
    return (mValue);
}
```

Zauważmy, że przed nazwą metody jest nazwa klasy z dwoma dwukropkami:

```
void SpreadsheetCell::setValue(double inValue)
```

Symbol `::` nazywany jest *operatorem zakresu*. W tym kontekście kompilator wie, że znajdująca się tu definicja metody `setValue()` jest częścią klasy `SpreadsheetCell`. Zauważmy jeszcze, że definiując metodę, nie używamy już określnika dostępu.

Dostęp do pól danych

Większość metod klasy, takich jak `setValue()` i `getValue()`, jest zawsze wykonywana na rzecz konkretnego obiektu klasy (wyjątkiem są metody *statyczne*, które omówimy dalej). W treści metody możemy sięgać do wszystkich pól *danego obiektu*. W powyższej definicji `setValue()` następująca instrukcja zmienia wartość zmiennej `mValue` znajdującej się wewnątrz obiektu:

```
mValue = inValue;
```

Jeśli metoda `setValue()` zostanie wywołana dla dwóch różnych obiektów, ten sam wiersz kodu (wykonany dwa razy, raz dla każdego obiektu) zmieni wartości zmiennych w dwóch różnych obiektach.

Wywoływanie innych metod

Z dowolnej metody klasy można wywoływać wszystkie inne metody tej klasy. Weźmy na przykład pod uwagę rozszerzenie klasy `SpreadsheetCell`. W prawdziwych arkuszach kalkulacyjnych komórka może zawierać zarówno liczbę, jak i tekst. Jeśli będziemy chcieli zinterpretować tekst komórki jako liczbę, arkusz spróbuje dokonać stosownej konwersji. Jeśli tekst nie jest poprawną liczbą, wartość komórki nie jest określana. W naszym programie łańcuchy tekstowe niebędące liczbami nadają komórce wartość 0. Oto pierwsze podejście do definicji klasy `SpreadsheetCell`, która może zawierać już tekst:

```

#include <string>
using std::string;

class SpreadsheetCell
{
public:
    void setValue(double inValue);
    double getValue();
    void setString(string inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};

```

Teraz nasza klasa może przechowywać dane jako liczby i jako tekst. Jeśli klient ustawi dane jako wartość `string`, wartość ta zostanie zamieniona na typ `double`, a typ `double` z powrotem na `string`. Jeśli tekst nie będzie prawidłową liczbą, pole `double` będzie równe 0. Pokazana definicja klasy zawiera dwie nowe metody do ustawiania i pobierania wartości tekstowej komórki oraz dwie nowe chronione *metody pomocnicze* konwertujące typ `double` na `string` i odwrotnie. Te metody pomocnicze korzystają ze strumieni łańcuchowych omawianych szczegółowo w rozdziale 14. Oto implementacja wszystkich metod:

```
#include "SpreadsheetCell.h"
```

```

#include <iostream>
#include <sstream>
using namespace std;

void SpreadsheetCell::setValue(double inValue)
{
    mValue = inValue;
    mString = doubleToString(mValue);
}

double SpreadsheetCell::getValue()
{
    return (mValue);
}

void SpreadsheetCell::setString(string inString)
{
    mString = inString;
    mValue = stringToDouble(mString);
}

string SpreadsheetCell::getString()
{
    return (mString);
}

```

```

string SpreadsheetCell::doubleToString(double inValue)
{
    ostringstream ostr;

    ostr << inValue;
    return (ostr.str());
}

double SpreadsheetCell::stringToDouble(string inString)
{
    double temp;

    istringstream istr(inString);

    istr >> temp;
    if (istr.fail() || !istr.eof()) {
        return (0);
    }
    return (temp);
}

```

Zauważmy, że każda metoda ustawiająca wartość robi konwersję. Dzięki temu `mValue` i `mString` zawsze mają prawidłowe wartości.

Wskaźnik `this`

Każde wywołanie normalnej metody przekazuje jako niejawny pierwszy parametr wskaźnik obiektu, na rzecz którego została wywołana; wskaźnik ten nazywany jest `this`. Można go używać do sięgania do pól danych lub wywoływania metod oraz można przekazywać do innych metod i funkcji. Czasami przydaje się on do unikania niejednoznaczności nazw. Moglibyśmy na przykład zdefiniować klasę `SpreadsheetCell` tak, aby metoda `setValue()` miała parametr `mValue`, a nie `inValue`. Wtedy metoda `setValue()` miałaby postać:

```

void SpreadsheetCell::setValue(double mValue)
{
    mValue = mValue;    // Niejednoznaczne!
    mString = doubleToString(mValue);
}

```

W tym wierszu jest błąd. O którą wartość `mValue` chodzi: przekazaną jako parametr czy o pole obiektu? Aby to wyjaśnić, można użyć wskaźnika `this`:

```

void SpreadsheetCell::setValue(double mValue)
{
    this->mValue = mValue;
    mString = doubleToString(mValue);
}

```

Jeśli jednak będziemy trzymać się konwencji nazewnicych opisanych w rozdziale 7., nigdy na takie problemy się nie natkniemy.

Wskaźnika `this` można też używać do wywoływania funkcji lub metody mającej za parametr wskaźnik obiektu z metody tegoż obiektu. Załóżmy na przykład, że piszemy samodzielną funkcję (nie metodę) `printCell()`:

```
void printCell(SpreadsheetCell* inCellp)
{
    cout << inCellp->getString() << endl;
}
```

Gdybyśmy chcieli wywołać `printCell()` z metody `setValue()`, musielibyśmy przekazać jako parametr wskaźnik `this`, czyli wskaźnik obiektu `SpreadsheetCell`, na którym działa metoda `setValue()`:

```
void SpreadsheetCell::setValue(double mValue)
{
    this->mValue = mValue;
    mString = doubleToString(this->mValue);
    printCell(this);
}
```

Użycie obiektów

Powyższa definicja klasy mówi, że `SpreadsheetCell` ma dwa pola, cztery metody publiczne oraz dwie metody chronione. Jednak definicja klasy nie powoduje utworzenia żadnego obiektu tej klasy; jest tylko i wyłącznie definicją. Z tego punktu widzenia klasa jest podobna do rysunków architektonicznych. Rysunki opisują dom, ale ich narysowanie nie jest równoznaczne ze zbudowaniem domu. Dom trzeba później zbudować właśnie na podstawie planów.

Tak samo w C++ można tworzyć obiekty klasy `SpreadsheetCell` na podstawie definicji tej klasy; wystarczy zadeklarować zmienną typu `SpreadsheetCell`. Tak jak budowniczowie mogą na podstawie jednych planów zbudować wiele domów, tak programista na podstawie jednej definicji klasy może utworzyć wiele obiektów. Obiekty można tworzyć na dwa sposoby: na stosie i na stercie.

Obiekty na stosie

Oto fragment kodu tworzący obiekty `SpreadsheetCell` na stosie i korzystający z nich:

```
SpreadsheetCell myCell, anotherCell;
myCell.setValue(6);
anotherCell.setValue(myCell.getValue());

cout << "komórka 1: " << myCell.getValue() << endl;
cout << "komórka 2: " << anotherCell.getValue() << endl;
```

Obiekty tworzy się, deklarując je jak każde inne zmienne, tyle tylko że zamiast nazwy typu zmiennej podaje się nazwę klasy. Kropka znajdująca się na przykład w zapisie `myCell.setValue(6)`; to operator „kropka”: pozwala on wywoływać metody obiektu. Gdyby istniały jakieś publiczne pola tego obiektu, można byłoby się do nich dostać także stosując operator „kropka”.

Wynikiem działania powyższego kodu jest:

```
komórka 1: 6
komórka 2: 6
```

Obiekty na stercie

Można też obiekty alokować na stercie za pomocą operatora `new`:

```
SpreadsheetCell* myCellp = new SpreadsheetCell();

myCellp->setValue(3.7);
cout << "komórka 1: " << myCellp->getValue() <<
    " " << myCellp->getString() << endl;
delete myCellp;
```

Kiedy tworzymy obiekt na stercie, jego metody wywołujemy i do pól sięgamy za pomocą operatora „strzałka”, `->`. Operator ten stanowi złożenie operatorów dereferencji (`*`) oraz dostępu do pola lub metody (`.`). Można byłoby użyć tych właśnie dwóch operatorów, ale zapis taki byłby bardzo nieczytelny:

```
SpreadsheetCell* myCellp = new SpreadsheetCell();
```

```
(*myCellp).setValue(3.7);
cout << "komórka 1: " << (*myCellp).getValue() <<
    " " << (*myCellp).getString() << endl;
delete myCellp;
```

Tak jak zwalnia się pamięć zaalokowaną na stercie na rozmaite zmienne, tak samo trzeba zwalniać pamięć zaalokowaną na obiekty, wywołując operator `delete`.

Jeśli alokujemy obiekt za pomocą operatora `new`, to kiedy przestaje nam być potrzebny, musimy zwolnić go operatorem `delete`.

Cykl życia obiektów

Cykl życia obiektu obejmuje trzy działania: tworzenie, niszczenie oraz przypisanie. Każdy obiekt jest tworzony, ale nie każdy jest niszczone i przypisywany. Trzeba dobrze zrozumieć, jak i kiedy obiekty są tworzone, niszczone i przypisywane oraz jak można te zachowania dostosowywać do swoich potrzeb.

Tworzenie obiektu

Obiekty tworzy się tam, gdzie się je deklaruje (o ile są one na stosie) lub przez jawną alokację pamięci na nie za pomocą operatora `new` lub `new[]`.

Często dobrze jest nadawać deklarowanym zmiennym wartości początkowe; na przykład liczby całkowite zwykle inicjalizuje się zerem:

```
int x = 0, y = 0;
```

Analogicznie należy inicjalizować obiekty. Można to zrobić deklarując i pisząc specjalną metodę nazywaną *konstruktorem*, która tworzy obiekt. Kiedy obiekt jest tworzony, wywołany jest jeden z jego konstruktorów.

Z czasami w angielskim tekście zamiast pełnego określenia konstruktora, *constructor*, można spotkać zapis skrócony, *ctor*.

Pisanie konstruktorów

Oto pierwsza próba dodania do klasy `SpreadsheetCell` konstruktora:

```
#include <string>
using std::string;

class SpreadsheetCell
{
public:
    SpreadsheetCell(double initialValue);
    void setValue(double inValue);
    double getValue();
    void setString(string inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};
```

Zauważmy, że konstruktor ma taką samą nazwę jak klasa i nie ma wartości zwracanej. Dotyczy to wszystkich konstruktorów. Tak jak trzeba podawać implementacje wszystkich normalnych metod, tak samo trzeba podać implementację konstruktora:

```
SpreadsheetCell::SpreadsheetCell(double initialValue)
{
    setValue(initialValue);
}
```

Konstruktor `SpreadsheetCell` to metoda klasy `SpreadsheetCell`, wobec czego C++ wymaga standardowego dodania zakresu, `SpreadsheetCell::`, przed nazwą metody. Sama nazwa metody to także `SpreadsheetCell`, więc ostatecznie otrzymujemy zabawnie wyglądające `SpreadsheetCell::SpreadsheetCell`. Nasza implementacja wywołuje `setValue()`, aby ustawić liczbowy i tekstowy zapis wartości komórki.

Użycie konstruktorów

Użycie konstruktorów powoduje utworzenie obiektu oraz inicjalizację jego wartości. Konstruktorów można używać zarówno do obiektów umieszczanych na stosie, jak i alokowanych na sterpie.

Konstruktory na stosie

Kiedy alokujemy obiekt `SpreadsheetCell` na stosie, używamy konstruktora następująco:

```
SpreadsheetCell myCell(5), anotherCell(4);
```

```
cout << "komórka 1: " << myCell.getValue() << endl;
cout << "komórka 2: " << anotherCell.getValue() << endl;
```

Zauważmy, że konstruktora `SpreadsheetCell` NIE wywołuje się jawnie. Nie można na przykład zapisać:

```
SpreadsheetCell myCell.SpreadsheetCell(5); //NIE DA SIĘ SKOMPILOWAĆ!
```

Tak samo nie można później wywołać konstruktora. Poniższy kod też jest błędny:

```
SpreadsheetCell myCell;
myCell.SpreadsheetCell(5); //NIE DA SIĘ SKOMPILOWAĆ!
```

Jedynym sposobem użycia konstruktora na stosie jest

```
SpreadsheetCell myCell(5);
```

Konstruktory na sterpie

Kiedy dynamicznie alokujemy obiekt `SpreadsheetCell`, używamy konstruktora następująco:

```
SpreadsheetCell* myCellp = new SpreadsheetCell(5);
SpreadsheetCell* anotherCellp;
anotherCellp = new SpreadsheetCell(4);
delete anotherCellp;
```

Zauważmy, że deklarujemy wskaźnik obiektu `SpreadsheetCell`, nie wywołując od razu konstruktora — inaczej niż na stosie, gdzie konstruktor jest wywoływany w chwili deklaracji zmiennej obiektowej.

Jak zawsze pamiętać trzeba o usunięciu za pomocą `delete` obiektów zaalokowanych operatorem `new`!

Definiowanie wielu konstruktorów

W klasie można podać więcej niż jeden konstruktor. Wszystkie konstruktory nazywają się tak samo (czyli tak jak klasa), ale mają inną liczbę parametrów lub inne typy tych parametrów.

W klasie `SpreadsheetCell` dobrze byłoby mieć dwa konstruktory: jeden pobierający wartość typu `double`, drugi typu `string`. Oto definicja klasy z drugim konstruktorem:

```

class SpreadsheetCell
{
public:
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(string initialValue);
    void setValue(double inValue);
    double getValue();
    void setString(string inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};

```

A teraz implementacja drugiego konstruktora:

```

SpreadsheetCell::SpreadsheetCell(string initialValue)
{
    setString(initialValue);
}

```

I kod wykorzystujący oba konstruktory:

```

SpreadsheetCell aThirdCell("test"); // konstruktor z par. string
SpreadsheetCell aFourthCell(4.4); // konstruktor z par. double
SpreadsheetCell* aThirdCellp = new SpreadsheetCell("4.4"); // konstr. string
cout << "aThirdCell: " << aThirdCell.getValue() << endl;
cout << "aFourthCell: " << aFourthCell.getValue() << endl;
cout << "aThirdCellp: " << aThirdCellp->getValue() << endl;
delete aThirdCellp;

```

Jeśli mamy wiele konstruktorów, chciałoby się zaimplementować jeden konstruktor za pomocą drugiego. Na przykład w konstruktorze z parametrem typu `string` moglibyśmy wywołać konstruktor z parametrem typu `double`:

```

SpreadsheetCell::SpreadsheetCell(string initialValue)
{
    SpreadsheetCell(stringToDouble(initialValue));
}

```

Wygląda niezłe. Ostatecznie przecież można wywołać jedne metody klasy z innych. Kod da się skompilować, skonsolidować i uruchomić, ale nie będzie działał zgodnie z oczekiwaniami. Jawne wywołanie konstruktora `SpreadsheetCell` tworzy nowy, tymczasowy obiekt bez nazwy typu `SpreadsheetCell`. Nie jest wywoływany konstruktor obiektu, który chcemy zainicjalizować.

Nie należy wywoływać jednego konstruktora danej klasy z innego.

Konstruktory domyślne

Konstruktor domyślny to konstruktor niemający parametrów; nazywany bywa też *konstruktorem zeroargumentowym*. Dzięki konstruktorowi domyślnemu można nadawać rozsądne wartości początkowe wszystkim polom, nawet jeśli klient ich nie poda.

Oto fragment definicji klasy `SpreadsheetCell` z konstruktorem domyślnym:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(string initialValue);
    // Reszta definicji klasy pominięta.
};
```

I oto pierwsza implementacja konstruktora domyślnego:

```
SpreadsheetCell::SpreadsheetCell()
{
    mValue = 0;
    mString = "";
}
```

Konstruktora domyślnego na stosie używa się następująco:

```
SpreadsheetCell myCell;
myCell.setValue(6);

cout << "komórka 1: " << myCell.getValue() << endl;
```

Powyższy kod utworzy nowy obiekt `myCell` klasy `SpreadsheetCell`, ustawi jego wartość oraz pokaże ją. W przeciwieństwie do innych konstruktorów, kiedy tworzymy obiekt na stosie, konstruktora domyślnego nie musimy wywoływać stosując składnię wywołania funkcji. Bazując na doświadczeniach z omawianymi wcześniej konstruktorami, chciałoby się być może napisać:

```
SpreadsheetCell myCell(); // ŹLE, choć skompiluje się
myCell.setValue(6); // Ten wiersz już się nie skompiluje.

cout << "komórka 1: " << myCell.getValue() << endl;
```

Niestety, wiersz, w którym próbujemy wywołać konstruktor domyślny, skompiluje się, a następny wiersz — już nie. Chodzi o to, że kompilator pierwszy wiersz traktuje jako deklarację funkcji o nazwie `myCell`, funkcji bezargumentowej, zwracającej obiekt typu `SpreadsheetCell`. Po przejściu do drugiego wiersza kompilator widzi próbę użycia nazwy funkcji jako obiektu.

Przy tworzeniu obiektów na stosie należy pomijać nawiasy w konstruktorach domyślnych.

Kiedy z kolei używamy konstruktora domyślnego na stercku, użycie składni funkcyjnej jest obowiązkowe:

```
SpreadsheetCell* myCellp = new SpreadsheetCell(); // Składnia funkcyjna.
```

Nie ma sensu rozważać, czemu C++ wymaga innej składni konstruktora domyślnego na stosie, a innej na stercie. Jest to jedna z tych ciekawostek, dzięki którym C++ jest tak ciekawym językiem.

Konstruktor domyślny generowany przez kompilator

Jeśli nasza klasa nie zawiera konstruktora domyślnego, nie można tworzyć obiektów tej klasy bez podania parametrów. Załóżmy na przykład, że mamy następującą definicję klasy SpreadsheetCell:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell(double initialValue);    // brak konstruktora domyślnego
    SpreadsheetCell(string initialValue);
    void setValue(double inValue);
    double getValue();
    void setString(string inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};
```

Jeśli mamy definicję taką jak powyżej, następujący kod nie da się skompilować:

```
SpreadsheetCell myCell;
myCell.setValue(6);
```

Ale ten kod przecież działa! Gdzie tu jest błąd? Otóż wcale nie ma błędu. Jednak nie zadeklarowaliśmy konstruktora domyślnego, więc nie możemy tworzyć obiektu, nie podając jego parametrów.

Tak naprawdę ważniejsze jest, dlaczego ten kod wcześniej działał. Otóż jeśli nie podamy *żadnych* konstruktorów, kompilator automatycznie wygeneruje jeden konstruktor bezparametrowy. Taki domyślny konstruktor wywołuje domyślne konstruktory wszystkich obiektów będących polami klasy, ale nie inicjalizuje pól typów elementarnych, na przykład typu `int` czy `double`. Tym niemniej wystarcza on do tworzenia obiektów danej klasy. Jeśli zadeklarujemy konstruktor domyślny lub jakikolwiek inny konstruktor, kompilator nie będzie generował żadnego konstruktora.

Pojęcie „konstruktor domyślny” nie oznacza tylko konstruktora generowanego automatycznie w sytuacji, kiedy sami nie zadeklarujemy żadnych innych konstruktorów. Konstruktor domyślny to inaczej konstruktor bezparametrowy.

Kiedy potrzebny jest konstruktor domyślny

Weźmy pod uwagę tablicę obiektów. Tworzenie takiej tablicy odbywa się w dwóch etapach: alokowany jest ciągły obszar pamięci na wszystkie obiekty, następnie dla każdego z tych obiektów jest wywoływany jego konstruktor domyślny. Język C++ nie ma żadnej konstrukcji składniowej, która pozwalałaby jawnie wywołać jakiegokolwiek inny konstruktor. Jeśli na przykład nie zdefiniujemy konstruktora domyślnego dla klasy `SpreadsheetCell`, poniższy kod nie da się skompilować:

```
SpreadsheetCell cells[3]; // BŁĄD KOMPILACJI - brak konstruktora domyślnego
SpreadsheetCell* myCellp = new SpreadsheetCell[10]; // także BŁĄD
```

Można tego problemu uniknąć w przypadku tablic na stosie, korzystając z wyrażeń *inicjalizujących*:

```
SpreadsheetCell cells[3] = {SpreadsheetCell(0), SpreadsheetCell(23),
    SpreadsheetCell(41)};
```

Jeśli jednak zamierzamy tworzyć tablice obiektów jakiejś klasy, zwykle łatwiej jest dodać w klasie konstruktor domyślny.

Konstruktory domyślne są też przydatne, kiedy chcemy tworzyć obiekty jakiejś klasy w innych klasach, co pokazano w następnym punkcie.

W końcu konstruktory domyślne są wygodnym mechanizmem w sytuacji, kiedy klasa jest klasą bazową dla całej hierarchii dziedziczenia. Wtedy podklasy mogą inicjalizować nadklasy, korzystając z ich konstruktorów domyślnych.

Listy inicjalizacyjne

Język C++ ma alternatywną metodę inicjalizacji pól w konstruktorze: *listę inicjalizacyjną*. Oto bezparametrowy konstruktor klasy `SpreadsheetCell` zmodyfikowany tak, aby używał tej składni:

```
SpreadsheetCell::SpreadsheetCell() : mValue(0), mString("")
{
}
```

Jak widać, lista inicjalizacyjna znajduje się między listą parametrów konstruktora a początkowym nawiasem klamrowym jego treści. Lista zaczyna się od dwukropka, dalej poszczególne jej elementy są rozdzielane przecinkami. Każdy element z listy stanowi inicjalizację pola w zapisie funkcyjnym lub w formie wywołania konstruktora nadklasy (więcej na ten temat w rozdziale 10.).

Inicjalizacja pól klasy za pomocą listy inicjalizacyjnej powoduje inne zachowanie niż inicjalizacja pól danych wewnątrz samego konstruktora. Kiedy C++ tworzy obiekt, musi utworzyć wszystkie pola danych w tym obiekcie, a dopiero wtedy wywołuje konstruktor. W ramach tworzenia tych pól musi wywołać konstruktory wszystkich obiektów będących polami; my później jedynie modyfikujemy ich wartość. Lista inicjalizatora pozwala podać wartości początkowe pól już na etapie ich tworzenia, co jest wydajniejsze od późniejszego przypisywania

im wartości. Co ciekawe, domyślna inicjalizacja obiektów typu `string` jest robiona ciągiem pustym, wobec czego jawna inicjalizacja `mString` pustym łańcuchem pokazana powyżej jest nadmiarowa.

Lista inicjalizacyjna pozwala na inicjalizację pól klasy już w chwili ich tworzenia.

Nawet jeśli nie zależy nam na wydajności, lista inicjalizacyjna jest przydatna, gdyż jak twierdzą niektórzy, kod z nią wygląda „czyściej”. Niektórzy wolą zwykłą składnię, z przypisywaniem wartości początkowych w treści konstruktora. Jednak niektóre rodzaje danych mogą być inicjalizowane tylko za pomocą listy; zestawiono je w poniższej tabeli.

Rodzaj danych	Wyjaśnienie
pola <code>const</code>	Z założenia nie można przypisywać wartości zmiennym zadeklarowanym ze słowem kluczowym <code>const</code> po ich utworzeniu. Wszelkie wartości muszą być podane już na etapie tworzenia takiej zmiennej.
pola będące referencjami	Referencje nie mogą istnieć same z siebie — zawsze muszą się do czegoś odwoływać.
pola będące obiektami bez konstruktora domyślnego	Język C++ próbuje inicjalizować pola będące obiektami za pomocą konstruktora domyślnego. Jeśli konstruktor taki nie istnieje, nie może takiego obiektu zainicjalizować.
nadklasy bez konstruktorów domyślnych	[omówione w rozdziale 10.]

Trzeba wiedzieć o jednej ważnej cesze list inicjalizacyjnych: powodują one inicjalizację pól w takiej kolejności, w jakiej te pola pojawiają się w definicji klasy; kolejność na samej liście nie ma znaczenia. Załóżmy na przykład, że ponownie przepiszemy konstruktor używający łańcucha klasy `SpreadsheetCell` i użyjemy listy inicjalizacyjnej następująco:

```
SpreadsheetCell::SpreadsheetCell(string initialValue) :
    mString(initialValue), mValue(stringToDouble(mString)) // NIEWŁAŚCIWA KOLEJNOŚĆ!
{
}
```

Kod się skompiluje (choć niektóre kompilatory zgłoszą ostrzeżenie), ale program nie będzie działał prawidłowo. Moglibyśmy zakładać, że pole `mString` zostanie zainicjalizowane przez `mValue`, gdyż `mString` znajduje się na liście inicjalizacyjnej wcześniej. Jednak w C++ tak nie jest. Klasa `SpreadsheetCell` ma deklarację `mValue` przed deklaracją `mString`:

```
class SpreadsheetCell
{
public:
    // pominięto część kodu
protected:
    // pominięto część kodu
    double mValue;
    string mString;
};
```

Wobec tego podczas inicjalizacji następuje próba zainicjalizowania `mValue` przed zainicjalizowaniem `mString`. Jednak `mValue` chce użyć wartości `mString` — jeszcze niezainicjalizowanej! W tym wypadku do inicjalizacji `mValue` należy na liście inicjalizacyjnej użyć parametru `initialValue` zamiast `mString`. Aby uniknąć pomyłek, należy też zamieścić kolejność elementów na liście inicjalizacyjnej:

```
SpreadsheetCell::SpreadsheetCell(string initialValue) :
    mValue(stringToDouble(initialValue)), mString(initialValue)
{
}
```

Lista inicjalizacyjna nie powoduje inicjalizacji pól w kolejności ich umieszczenia na tej liście, ale w kolejności ich deklarowania w definicji klasy.

Konstruktory kopiujące

Istnieje w C++ specjalny konstruktor, *konstruktor kopiujący*, który umożliwia tworzenie obiektów będących dokładnymi kopiami innych obiektów. Jeśli sami nie napiszemy konstruktora kopiującego, C++ generuje go dla nas, inicjalizując pola nowego obiektu odpowiednimi polami obiektu źródłowego. W przypadku pól będących obiektami inicjalizacja oznacza wywołanie ich konstruktorów kopiujących.

Oto deklaracja konstruktora kopiującego klasy `SpreadsheetCell`:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(string initialValue);
    SpreadsheetCell(const SpreadsheetCell &src);
    void setValue(double inValue);
    double getValue();
    void setString(string inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(string inString);

    double mValue;
    string mString;
};
```

Konstruktor kopiujący pobiera referencję stałą (z modyfikatorem `const`). Tak jak inne konstruktory, konstruktor kopiujący nie zwraca wartości. W tym konstruktorze należy skopiować wszystkie pola danych obiektu źródłowego. Formalnie można oczywiście zrobić wszystko, co się robi w zwykłych konstruktorach, ale dobrą zasadą jest realizowanie zachowania standardowego i inicjalizowanie nowych obiektów kopiami starego. Oto przykładowa implementacja konstruktora kopiującego klasy `SpreadsheetCell`:

```

SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell &src) :
    mValue(src.mValue), mString(src.mString)
{
}

```

Należy zwrócić uwagę na listę inicjalizacyjną. Różnicę między ustawianiem wartości w tej liście a w treści konstruktora omówimy dalej, kiedy będziemy omawiać przypisanie.

Generowany przez kompilator konstruktor kopiujący klasy `SpreadsheetCell` jest identyczny jak powyższy. Wobec tego w celu uproszczenia moglibyśmy w ogóle go pominąć i polegać na tym tworzonym przez kompilator. W rozdziale 10. opiszemy klasy, w których konstruktor kopiujący generowany przez kompilator już nie wystarcza.

Kiedy wywoływany jest konstruktor kopiujący

Domyślnie w języku C++ parametry do funkcji są przekazywane przez wartość. Oznacza to, że funkcja lub metoda nie otrzymuje samej zmiennej, ale jej kopię. Wobec tego kiedy do funkcji lub metody przekazujemy obiekt, kompilator wywołuje konstruktor kopiujący tworzący nowy obiekt i inicjalizujący go.

Przypomnijmy definicję metody `setString()` z klasy `SpreadsheetCell`:

```

void SpreadsheetCell::setString(string inString)
{
    mString = inString;
    mValue = stringToDouble(mString);
}

```

Przypomnijmy jeszcze, że w C++ `string` jest klasą, a nie typem wbudowanym. Kiedy w kodzie wywołujemy metodę `setString()`, przekazując jej parametr będący łańcuchem, parametr `inString` jest inicjalizowany przez wywołanie jego konstruktora kopiującego. Parametrem konstruktora kopiującego jest łańcuch przekazany metodzie `setString()`. Poniżej konstruktor kopiujący klasy `string` jest wykonywany na rzecz obiektu `inString` w metodzie `setString()`, jako parametr przekazywana jest mu zmienna `name`:

```

SpreadsheetCell myCell;
string name = "nagiłówek 1";

myCell.setString(name); // kopiuje obiekt name

```

Kiedy metoda `setString()` kończy swoje działanie, obiekt `inString` jest usuwany. Był on jedynie kopią `name`, więc sama zmienna `name` pozostaje nienaruszona.

Konstruktor kopiujący jest też wywoływany, kiedy zwracamy z funkcji lub metody obiekt. W tym wypadku kompilator tworzy tymczasowy obiekt bez nazwy, korzystając z konstruktora kopiującego. W rozdziale 17. omówimy dokładniej znaczenie obiektów tymczasowych.

Jawne wywoływanie konstruktora kopiującego

Konstruktora kopiującego można używać jawnie. Często przydaje się to do tworzenia nowego obiektu będącego kopią innego obiektu. Na przykład kopie obiektów `SpreadsheetCell` możemy tworzyć następująco:

```
SpreadsheetCell myCell12(4);
SpreadsheetCell anotherCell(myCell12); // anotherCell ma teraz wartości z myCell2
```

Przekazywanie obiektów przez referencję

Aby uniknąć kopiowania obiektów przy przekazywaniu ich do funkcji i metod, można zadeklarować parametr funkcji czy metody jako referencję do obiektu. Przekazywanie obiektów przez referencję jest zwykle wydajniejsze od przekazywania ich przez wartość, gdyż wystarczy skopiować adres, a nie całą zawartość obiektu. Poza tym dzięki przekazywaniu przez referencję unika się problemów z dynamicznym alokowaniem pamięci, czym zajmujemy się w rozdziale 9.

Zamiast przekazywać obiekty przez wartość, lepiej przekazać je przez referencję z modyfikatorem `const`.

Kiedy przekazujemy obiekt przez referencję, otrzymująca taki parametr funkcja lub metoda może zmienić wartości w przekazanym obiekcie. Jeśli obiekt przekazujemy przez referencję jedynie z uwagi na wydajność, powinniśmy do parametru dodać modyfikator `const`. Oto definicja klasy `SpreadsheetCell`, gdzie obiekty `string` są przekazywane jako referencje stałe:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell &src);
    void setValue(double inValue);
    double getValue();
    void setString(const string& inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(const string& inString);

    double mValue;
    string mString;
};
```

A to implementacja metody `setString()`. Zauważmy, że treść tej metody nie zmienia się; inny jest jedynie typ parametru:

```
void SpreadsheetCell::setString(const string& inString)
{
    mString = inString;
    mValue = stringToDouble(mString);
}
```

Metody `SpreadsheetCell` zwracające obiekt `string` też zwracają go przez wartość. Zwrócenie referencji do pola jest ryzykowne, gdyż referencja ta będzie poprawna tylko tak długo, jak długo obiekt będzie „żywy”. Kiedy obiekt zostanie usunięty z pamięci, referencja będzie nieprawidłowa. Czasami jednak istnieją powody, aby zwracać referencje do pól obiektu; zajmijmy się tym dalej w tym rozdziale i w rozdziałach następujących.

Podsumowanie informacji o konstruktorach generowanych przez kompilator

Kompilator automatycznie generuje bezparametrowy konstruktor oraz konstruktor kopiujący dla każdej klasy. Konstruktory definiowane przez programistę zastępują konstruktory kompilatora zgodnie z następującymi zasadami:

Jeśli zdefiniujemy...	...to kompilator wygeneruje...	...i możemy tworzyć obiekt...	Przykład
[brak konstruktorów]	konstruktor bezparametrowy konstruktor kopiujący	bez parametrów; jako kopię innego obiektu	<code>SpreadsheetCell cell;</code> <code>SpreadsheetCell myCell(cell);</code>
tylko konstruktor bezparametrowy	konstruktor kopiujący	bez parametrów jako kopię innego obiektu	<code>SpreadsheetCell cell;</code> <code>SpreadsheetCell myCell(cell);</code>
tylko konstruktor kopiujący	nic	teoretycznie jako kopię innego obiektu; praktycznie nie można stworzyć żadnych obiektów	Nie ma przykładów.
tylko konstruktor jedno- lub wieloargumentowy (niekopiujący)	konstruktor kopiujący	z parametrami jako kopię innego obiektu	<code>SpreadsheetCell cell(6);</code> <code>SpreadsheetCell myCell(cell);</code>
konstruktor bezparametrowy oraz jedno- lub wieloparametrowy konstruktor (niekopiujący)	konstruktor kopiujący	bez parametrów z parametrami jako kopię innego obiektu	<code>SpreadsheetCell cell;</code> <code>SpreadsheetCell myCell(5);</code> <code>SpreadsheetCell anotherCell(cell);</code>

Zwróćmy uwagę na brak symetrii między konstruktorem domyślnym a konstruktorem kopiującym. Póki nie zdefiniujemy jawnie konstruktora kopiującego, kompilator będzie go tworzył. Jeśli jednak zdefiniujemy *dowolny* konstruktor, kompilator przestaje generować konstruktor domyślny.

Usuwanie obiektu

Kiedy usuwany jest obiekt, zachodzą dwa zdarzenia: wywoływana jest metoda obiektu nazywana *destruktor* oraz zwalniana jest pamięć zaalokowana na ten obiekt. Destruktor umożliwia zrobienie porządków w obiekcie, na przykład zwolnienie zaalokowanej dynamicznie

pamięci czy zamknięcie otwartych plików. Jeśli nie zadeklarujemy destruktora, kompilator sam się tym zajmie — wygeneruje destruktor rekurencyjnie niszczący (usuwający) obiekty będące polami bieżącej klasy. O destruktorach będziemy pisali w rozdziale 9., przy omawianiu dynamicznej alokacji pamięci.

Obiekty znajdujące się na stosie są usuwane, kiedy wychodzą *poza zakres*, czyli kiedy kończy się wykonywanie bieżącej funkcji, metody lub innego *bloku*. Innymi słowy, kiedy w kodzie dochodzimy do zamykającego nawiasu klamrowego, wszystkie obiekty utworzone na stosie między tym nawiasem a pasującym do niego nawiasem otwierającym są usuwane. Można to obejrzeć w poniższym programie:

```
int main(int argc, char** argv)
{
    SpreadsheetCell myCell(5);

    if (myCell.getValue() == 5) {
        SpreadsheetCell anotherCell(6);
    } // koniec bloku - usuwany jest obiekt anotherCell

    cout << "myCell: " << myCell.getValue() << endl;

    return (0);
} // koniec bloku - usuwany jest obiekt myCell
```

Obiekty tworzone na stosie są usuwane w kolejności odwrotnej, niż były deklarowane (zatem i tworzone). Na przykład w poniższym fragmencie kodu obiekt `myCell12` jest tworzony przed `anotherCell12`, więc `anotherCell12` będzie usunięty wcześniej niż `myCell12` (zwróćmy uwagę, że nowy blok możemy utworzyć w dowolnym miejscu kodu, wstawiając tam otwierający nawias klamrowy):

```
{
    SpreadsheetCell myCell12(4);
    SpreadsheetCell anotherCell12(5); // myCell12 jest tworzony przed anotherCell12
} // anotherCell12 jest usuwany przed usunięciem myCell12
```

Ta sama kolejność obowiązuje także w przypadku obiektów będących polami innych obiektów. Przypomnijmy, że pola są inicjalizowane w kolejności ich deklaracji w klasie. Wobec tego zgodnie z zasadą usuwania obiektów w kolejności odwrotnej do kolejności ich tworzenia pola klasy będą usuwane w kolejności odwrotnej do kolejności ich deklaracji.

Obiekty alokowane na stercie nie są automatycznie usuwane; trzeba wywołać `delete` na odnoszącym się do nich wskaźniku, wtedy zostanie wywołany destruktor i zwolniona pamięć. Pokazano to w poniższym przykładzie:

```
int main(int argc, char** argv)
{
    SpreadsheetCell* cellPtr1 = new SpreadsheetCell(5);
    SpreadsheetCell* cellPtr2 = new SpreadsheetCell(6);

    cout << "cellPtr1: " << cellPtr1->getValue() << endl;

    delete cellPtr1; // usuwa cellPtr1

    return (0);
} // cellPtr2 NIE jest usuwany, gdyż nie wywołano na nim delete
```

Przypisania do obiektów

Tak jak można w C++ przypisać wartość jednej zmiennej typu `int` innej zmiennej, tak samo można przypisać jeden obiekt do innego. Na przykład poniżej przypisujemy wartość obiektu `myCell` obiektowi `anotherCell`:

```
SpreadsheetCell myCell(5), anotherCell;
```

```
SpreadsheetCell = myCell;
```

Chciałoby się powiedzieć, że `myCell` jest kopiowany do zmiennej `anotherCell`. Jednak w C++ kopiowanie ma miejsce tylko wtedy, kiedy obiekt jest inicjalizowany. Jeśli obiekt ma już wartość, która jest nadpisywana, mówimy o *przypisaniu*. W C++ mamy konstruktory kopiujące; skoro konstruktory, to znaczy funkcje używane do tworzenia obiektu, a nie do robienia później przypisań obiektu istniejącego.

Wobec tego w C++ istnieje inna metoda w każdej klasie, która pozwala robić przypisania obiektów tej klasy: metodę tę nazywamy *operatorem przypisania*. Nazwa tej metody to `operator=`, gdyż tak naprawdę jest to przeciążenie operatora `=` na rzecz tej klasy. W powyższym przykładzie wywoływany jest operator przypisania obiektu `anotherCell`, jego parametrem jest `myCell`.

Jak zwykle, jeśli nie napiszemy własnego operatora przypisania, kompilator go za nas wygeneruje. Domyślne przypisanie działa tak, jak domyślne kopiowanie: rekurencyjnie przypisuje każdemu polu odpowiadające mu pole z obiektu źródłowego. Składnia jest jednak nieco nienaturalna.

Deklarowanie operatora przypisania

Oto kolejna definicja klasy `SpreadsheetCell`, tym razem z operatorem przypisania:

```
class SpreadsheetCell
{
public:
    SpreadsheetCell();
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(const string& initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
    void setValue(double inValue);
    double getValue();
    void setString(const string& inString);
    string getString();

protected:
    string doubleToString(double inValue);
    double stringToDouble(const string& inString);

    double mValue;
    string mString;
};
```

Operator przypisania, podobnie jak konstruktor kopiujący, ma referencję stałą do obiektu źródłowego. W tym wypadku obiektem źródłowym jest `rhs`, skrót od angielskiego *right-hand-side*, czyli „prawa strona”; chodzi oczywiście o prawą stronę znaku `=`. Obiekt, na rzecz którego wywoływany jest operator przypisania, w wyrażeniu znajduje się po lewej stronie operatora przypisania.

Operator przypisania, w przeciwieństwie do konstruktora kopiującego, zwraca referencję do obiektu `SpreadsheetCell`. Wynika to stąd, że można tworzyć łańcuchy przypisań, jak poniżej:

```
myCell = anotherCell = aThirdCell;
```

Kiedy wykonamy taką instrukcję, najpierw zadziała operator przypisania obiektu `anotherCell` z `aThirdCell` jako parametrem. Następnie zadziała operator przypisania `myCell`, ale jego parametrem nie będzie `anotherCell`; parametrem będzie *wynik* przypisania `aThirdCell` do `anotherCell`. Jeśli przypisanie nie zwróciłoby wyniku, nie byłoby czego przekazać do `myCell`!

Zastanówmy się, czemu operator przypisania obiektu `myCell` nie skorzysta po prostu z `anotherCell`. Wynika to stąd, że znak równości to tak naprawdę tylko skrócony zapis wywołania metody. Kiedy spojrzymy na pełną składnię powyższego kodu, wszystko stanie się jasne:

```
myCell.operator=(anotherCell.operator=(aThirdCell));
```

Teraz widzimy, że wywołanie `operator=` dla obiektu `anotherCell` musi zwrócić wartość, która zostanie przekazana do metody `operator=` obiektu `myCell`. Prawidłowa wartość, jaka zostanie zwrócona, to sam obiekt `anotherCell`, gdyż to on może być źródłem przypisania do `myCell`. Jednak bezpośrednie zwracanie obiektu `anotherCell` byłoby niewygodne, dlatego zwracana jest referencja do tego obiektu.

Można byłoby zadeklarować operator przypisania tak, aby zwracał wartość dowolnego interesującego nas typu, w tym także `void`. Jednak mimo to zawsze powinniśmy zwracać referencję obiektu, gdyż tego właśnie oczekuje klient.

Definiowanie operatora przypisania

Implementacja operatora przypisania jest podobna do definicji konstruktora kopiującego, jest jednak kilka ważnych różnic. Po pierwsze, konstruktor kopiujący jest wywoływany w celu inicjalizacji, zatem obiekt docelowy nie zawiera jeszcze poprawnych wartości. Operator przypisania może nadpisać dotychczasowe wartości znajdujące się w obiekcie. Nie jest to istotne tak długo, jak długo nie alokujemy dynamicznie pamięci — więcej na ten temat w rozdziale 10.

Po drugie, w C++ można przypisać obiekt do samego siebie. Poniższy kod jest zatem poprawny:

```
SpreadsheetCell cell(4);
cell = cell; //przypisanie do samego siebie
```

Definiowany przez nas operator przypisania nie powinien uniemożliwiać przypisania wartości obiektu do samego siebie, ale nie powinien też w takiej sytuacji przypisywać kolejnych wartości. Wobec tego w operatorze powinniśmy od razu sprawdzić, czy mamy do czynienia z przypisaniem do samego siebie i jeśli tak, zakończyć działanie metody.

Oto definicja operatora przypisania klasy SpreadsheetCell:

```
SpreadsheetCell& SpreadsheetCell::operator=(const SpreadsheetCell& rhs)
{
    if (this == &rhs) {
```

Sprawdzamy tutaj, czy zachodzi przypisanie obiektu do samego siebie. Dzieje się tak wtedy, kiedy lewa i prawa strona są takie same. Jeden ze sposobów sprawdzenia, czy dwa obiekty są tak naprawdę tym samym obiektem, to sprawdzenie, czy zajmują w pamięci to samo miejsce, czyli czy równe są wskaźniki na nie. Wskaźnik `this`, jak pamiętamy, to wskaźnik obiektu dostępny w dowolnej metodzie zdefiniowanej w tym obiekcie. Zatem `this` jest to wskaźnik lewej strony przypisania. Z kolei `&rhs` to wskaźnik obiektu z prawej strony przypisania. Jeśli wskaźniki te są sobie równe, mamy do czynienia z przypisaniem obiektu do samego siebie.

```
        return (*this);
    }
```

Wskaźnik obiektu, którego operator przypisania jest wykonywany, to `this`, zatem `*this` to sam obiekt. Kompilator zwróci referencję obiektu, aby zwracana wartość była zgodna z deklaracją.

```
        mValue = rhs.mValue;
        mString = rhs.mString;
```

Powyżej mamy kopiowanie wartości.

```
        return (*this);
    }
```

I na koniec zwracamy `*this`, jak to przed chwilą opisywaliśmy.

Składnia nadpisywania metody `operator=` może wydawać się nieco dziwna. Tak to już jest przy nauce C i C++ — po prostu niektóre konstrukcje, jak choćby instrukcja `switch`, wydają się w pierwszej chwili nienaturalne. W przypadku metody `operator=` dochodzimy do bardzo fundamentalnych cech języka: zmieniamy znaczenie operatora `=`. Niestety, aby mieć tak duże możliwości, trzeba korzystać z dość niezwykłej składni. Ale nic to, można się przyzwyczaić!

Odróżnianie kopiowania od przypisania

Czasami trudno jest powiedzieć, kiedy obiekty są inicjalizowane konstruktorem kopiującym, a kiedy przypisywane operatorem przypisania. Spójrzmy na poniższy kod:

```
SpreadsheetCell myCell(5);
SpreadsheetCell anotherCell(myCell);
```

Obiekt `AnotherCell` jest tworzony za pomocą konstruktora kopiującego.

```
SpreadsheetCell aThirdCell = myCell;
```

Obiekt `aThirdCell` także jest tworzony za pomocą konstruktora kopiującego; w tym wierszu metoda `operator=` w ogóle nie jest wywoływana! Pokazana składnia oznacza tak naprawdę `SpreadsheetCell aThirdCell(myCell);`.

```
anotherCell = myCell; //wywołuje operator= dla anotherCell
```

Tym razem obiekt `anotherCell` był już wcześniej utworzony, więc kompilator wywołuje metodę `operator=`.

Znak równości, `=`, nie zawsze oznacza przypisanie! Jeśli występuje wraz z deklaracją zmiennej, może to być też skrótowy zapis tworzenia obiektu przez kopiowanie.

Obiekty jako wartości zwracane

Kiedy zwracamy z funkcji lub metod obiekty, czasami trudno dojść, jakie są robione przypisania i kopiowania. Przypomnijmy postać metody `getString()`:

```
string SpreadsheetCell::getString()
{
    return (mString);
}
```

Teraz spójrzmy na następujący kod:

```
SpreadsheetCell myCell12(5);
string s1;
s1 = myCell12.getString();
```

Kiedy metoda `getString()` zwraca `mString`, kompilator tak naprawdę tworzy tymczasowy obiekt `string` bez nazwy, wywołując konstruktor kopiujący klasy `string`. Kiedy przypisujemy wynik zmiennej `s1`, wywoływany jest operator przypisania obiektu `s1`, jego parametrem jest tymczasowy obiekt `string`. Następnie obiekt tymczasowy jest usuwany. Wobec tego w tym jednym wierszu kodu następuje wywołanie konstruktora kopiującego oraz dla dwóch różnych obiektów operatora przypisania.

Jeśli ktoś się jeszcze nie pogubił, oto następny przykład:

```
SpreadsheetCell myCell13(5);
string s2 = myCell13.getString();
```

Tym razem metoda `getString()` tworzy na zwracaną wartość tymczasowy obiekt `string` bez nazwy. Jednak teraz w `s1` wywoływany jest konstruktor kopiujący, a nie operator przypisania.

Jeśli kiedykolwiek zapomnimy, w jakiej kolejności odbywają się poszczególne wywołania albo który konstruktor czy operator jest wywoływany, łatwo to sprawdzić wyświetlając z kodu dodatkowe komunikaty albo korzystając z programu uruchomieniowego (*debugera*).

Konstruktory kopiujące a pola obiektów

Trzeba jeszcze zwrócić uwagę na różnicę między przypisaniem a wywołaniem konstruktorów kopiujących w innych konstruktorach. Jeśli obiekt zawiera inne obiekty, generowane przez kompilator konstruktory kopiujące wywołują konstruktory kopiujące wszystkich takich obiektów wewnętrznych, a następnie rekurencyjnie dalej. Kiedy piszemy własny konstruktor kopiujący, możemy to samo zrealizować stosując listę inicjalizacyjną, czym zajmowaliśmy się wcześniej. Jeśli pominiemy na tej liście jakieś pole klasy, przed wykonaniem treści naszego konstruktora kompilator wywoła domyślną inicjalizację takiego pola (czyli w przypadku obiektów wywoła bezparametrowy konstruktor). Kiedy wykonywany jest kod konstruktora, wszystkie pola danych będące obiektami już są zainicjalizowane.

Moglibyśmy napisać nasz konstruktor kopiujący, nie korzystając z listy inicjalizującej:

```
SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src)
{
    mValue = src.mValue;
    mString = src.mString;
}
```

Jednak kiedy przypisujemy wartości do pól w konstruktorze kopiującym, korzystamy z operatora przypisania, a nie konstruktorów kopiujących, gdyż pola są już zainicjalizowane.

Podsumowanie

W tym rozdziale powiedzieliśmy o najważniejszych obiektowych mechanizmach C++: klasach i obiektach. Omówiliśmy nieco składnię pozwalającą definiować klasy i używać obiektów, w tym powiedzieliśmy o kontroli dostępu. Następnie omówiliśmy cykl życia obiektu: jego tworzenie, usuwanie i przypisywanie, powiedzieliśmy też, kiedy jakie metody są wywoływane. Powiedzieliśmy więcej o składni konstruktorów, w tym o listach inicjalizacyjnych. Wiemy już, jakie konstruktory generuje dla nas kompilator, w jakich warunkach. Wiemy, że konstruktory domyślne są bezparametrowe.

Dla części Czytelników ten rozdział stanowił jedynie przypomnienie. Innym — miejmy nadzieję — otworzył oczy na programowanie obiektowe w C++. Tak czy inaczej, teraz już potrafimy sprawnie posługiwać się obiektami i klasami, więc w rozdziale 9. poznamy więcej szczegółów i sztuczek.