



Nowoczesne projektowanie w C++

Uogólnione implementacje wzorców projektowych

KORZYSTAJ Z NOWOCZESNYCH TECHNIK W C++!

- Jak korzystać z wzorców projektowych w C++?
- Jak stworzyć dokładnie jedną instancję obiektu?
- Jak używać inteligentnych wskaźników?

Andrei Alexandrescu

» Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

» 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

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2011

Nowoczesne projektowanie w C++. Uogólnione implementacje wzorców projektowych

Autor: Andrei Alexandrescu

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-246-3301-2

Tytuł oryginału: [Modern C++ Design: Generic Programming and Design Patterns Applied](#)

Format: 172×245, stron: 352



Korzystaj z nowoczesnych technik w C++!

- Jak korzystać z wzorców projektowych w C++?
- Jak stworzyć dokładnie jedną instancję obiektu?
- Jak używać inteligentnych wskaźników?

Język C++ jest obecny na rynku już niemal trzydzieści lat, a jednak nadal świetnie spełnia swoje zadania. Jest powszechnie używany, a wręcz niezastąpiony w wielu dziedzinach programowania. Wszędzie tam, gdzie potrzebna jest najwyższa wydajność oraz pełna kontrola nad zasobami i przebiegiem programu, sprawdza się wyśmienicie. Wystarczy odrobina chęci, dobry podręcznik i trochę czasu, aby wykorzystać pełną moc C++ w nowoczesnych technikach programowania.

Książkę, która Ci w tym pomoże, trzymasz właśnie w rękach. Czy znajdziesz czas i ochotę, aby zgłębić zawartą w niej wiedzę? Gwarantujemy, że warto! W trakcie lektury dowiesz się, jak zaimplementować w C++ najpopularniejsze wzorce projektowe. Dzięki nim błyskawicznie oprogramujesz typowe rozwiązania. Nauczysz się tworzyć dokładnie jedną instancję obiektu oraz zobaczysz, jak korzystać z fabryki obiektów czy inteligentnych wskaźników. Ponadto zapoznasz się z technikami projektowania klas, asercjami w trakcie kompilacji oraz uogólnionymi funktorami. Dzięki tej książce poczujesz na nowo satysfakcję z pisania programów w języku C++!

- Projektowanie klas
- Asercje czasu kompilacji
- Listy typów
- Alokowanie małych obiektów
- Funktory uogólnione
- Inteligentne wskaźniki
- Fabryka obiektów i fabryka abstrakcyjna
- Tworzenie dokładnie jednego obiektu – wzorzec singleton
- Multimetry

Czerp satysfakcję z korzystania z nowoczesnych technik programowania w C++!

Spis treści

Przedmowy	9
Wstęp	13
Podziękowania	19

I	
Techniki	21

1.

Klasy konfigurowane wytycznymi	23
1.1. Projektowanie oprogramowania — kłęska urodzaju?	23
1.2. Porażka interfejsu „wszechstronnego”	24
1.3. Ratunek w wielodziedziczeniu?	26
1.4. Zalety szablonów	26
1.5. Wytyczne i klasy wytycznych	28
1.6. Wytyczne rozszerzone	32
1.7. Destruktory klas wytycznych	33
1.8. Elementy opcjonalne w konkretyzacji częściowej	34
1.9. Łączenie klas wytycznych	35
1.10. Klasy wytycznych a konfigurowanie struktury	37
1.11. Wytyczne zgodne i niezgodne	38
1.12. Dekompozycja klas na wytyczne	40
1.13. Podsumowanie	42

2.

Techniki	43
2.1. Asercje statyczne	44
2.2. Częściowa specjalizacja szablonu	46
2.3. Klasy lokalne	48
2.4. Mapowanie stałych na typy	49

2.5. Odwzorowanie typu na typ	52
2.6. Wybór typu	53
2.7. Statyczne wykrywanie dziedziczenia i możliwości konwersji	55
2.8. TypeInfo	58
2.9. NullType i EmptyType	60
2.10. Cechy typów	61
2.11. Podsumowanie	68

3.

Listy typów	71
3.1. Listy typów — do czego?	71
3.2. Definiowanie list typów	73
3.3. Liniowe tworzenie list typów	75
3.4. Obliczanie długości listy	76
3.5. Przygrywka	77
3.6. Dostęp swobodny (indeksowany)	77
3.7. Przeszukiwanie list typów	79
3.8. Dopisywanie do listy typów	80
3.9. Usuwanie typu z listy	81
3.10. Usuwanie duplikatów	82
3.11. Zastępowanie elementu na liście typów	83
3.12. Częściowe porządkowanie listy typów	84
3.13. Generowanie klas z list typów	87
3.14. Podsumowanie	97
3.15. Listy typów — na skróty	97

4.

Przydział pamięci dla niewielkich obiektów	101
4.1. Domyślny alokator pamięci dynamicznej	102
4.2. Zasada działania alokatora pamięci	102
4.3. Alokator małych obiektów	104
4.4. Chunk	105
4.5. Alokator przydziałów o stałym rozmiarze	108
4.6. Klasa SmallObjAllocator	112
4.7. 3 x Tak	113
4.8. Proste, skomplikowane, a potem znów proste	116
4.9. Konfigurowanie alokatora	117
4.10. Podsumowanie	118
4.11. Alokator małych obiektów — na skróty	119

II

Komponenty	121
------------------	-----

5.

Uogólnione obiekty funkcyjne	123
5.1. Wzorzec Command	124
5.2. Wzorzec Command w praktyce	126

5.3. Byty funkcyjne w C++	127
5.4. Szkielet szablonu klasy Functor	129
5.5. Delegowanie wywołania Functor::operator()	133
5.6. Funktory	135
5.7. Raz a dobrze	137
5.8. Konwersje typów argumentów i wartości zwracanej	139
5.9. Wskaźniki do metod	140
5.10. Wiązanie argumentów wywołania	144
5.11. Żądania skomasowane	147
5.12. Z życia wzięte (1) — duży koszt delegacji funkcji	147
5.13. Z życia wzięte (2) — alokacje na stercie	149
5.14. Szablon Functor w implementacji cofnij-powtórz	150
5.15. Podsumowanie	151
5.16. Functor — na skróty	152

6.

Singletony	155
6.1. Statyczne dane i statyczne funkcje nie czynią singletona	156
6.2. Podstawowe idiomy C++ dla singletonów	157
6.3. Wymuszanie unikalności	158
6.4. Usuwanie singletona	159
6.5. Problem martwych referencji	162
6.6. Problem martwych referencji (1) — singleton à la feniks	164
6.7. Problem martwych referencji (2) — sterowanie żywotnością	167
6.8. Implementowanie singletonów z żywotnością	169
6.9. Życie w wielu wątkach	173
6.10. Podsumowanie doświadczeń	176
6.11. Stosowanie szablonu SingletonHolder	181
6.12. Podsumowanie	183
6.13. Szablon klasy SingletonHolder — na skróty	183

7.

Inteligentne wskaźniki	185
7.1. Elementarz inteligentnych wskaźników	186
7.2. Oferta	187
7.3. Przydział obiektów inteligentnych wskaźników	188
7.4. Metody inteligentnego wskaźnika	190
7.5. Strategie zarządzania posiadaniem	191
7.6. Operator pobrania adresu	199
7.7. Niejawna konwersja na typ gołego wskaźnika	200
7.8. Równość i różność	202
7.9. Porównania porządkujące	207
7.10. Kontrola i raportowanie o błędach	210
7.11. Wskaźniki niemodyfikowalne i wskaźniki do wartości niemodyfikowalnych	211
7.12. Tablice	212
7.13. Inteligentne wskaźniki a wielowątkowość	213
7.14. Podsumowanie doświadczeń	217
7.15. Podsumowanie	223
7.16. Wskaźniki SmartPtr — na skróty	224

8.

Wytwórnice obiektów	225
8.1. Przydatność wytwórni obiektów	226
8.2. Wytwórnice obiektów w C++ — klasy a obiekty	228
8.3. Implementowanie wytwórni obiektów	229
8.4. Identyfikatory typu	234
8.5. Uogólnienie	235
8.6. Sprostowania	239
8.7. Wytwórnice klonów	240
8.8. Stosowanie wytwórni obiektów z innymi komponentami uogólnionymi	243
8.9. Podsumowanie	244
8.10. Szablon klasy Factory — na skróty	244
8.11. Szablon klasy CloneFactory — na skróty	245

9.

Wzorzec Abstract Factory	247
9.1. Rola wytwórni abstrakcyjnej w architekturze	247
9.2. Interfejs uogólnionej wytwórni abstrakcyjnej	251
9.3. Implementacja AbstractFactory	254
9.4. Implementacja wytwórni abstrakcyjnej z prototypowaniem	256
9.5. Podsumowanie	261
9.6. Szablony AbstractFactory i ConcreteFactory — na skróty	262

10.

Wzorzec Visitor	265
10.1. Elementarz modelu wizytacji	265
10.2. Przeciżenia i metoda wychwytyjąca	271
10.3. Wizytacja acykliczna	273
10.4. Uogólniona implementacja wzorca Visitor	278
10.5. Powrót do wizytacji „cyklicznej”	284
10.6. Wariacje	287
10.7. Podsumowanie	290
10.8. Uogólnione komponenty wzorca wizytacji — na skróty	290

11.

Wielometody	293
11.1. Czym są wielometody?	294
11.2. Przydatność wielometod	295
11.3. Podwójne przełączanie — metoda siłowa	296
11.4. Metoda siłowa w wersji automatyzowanej	298
11.5. Symetria rozprowadzania metodą siłową	303
11.6. Rozprowadzanie logarymiczne	307
11.7. Symetria w FnDispatcher	312
11.8. Podwójne rozprowadzanie do funkcyj	313
11.9. Konwertowanie argumentów — statyczne czy dynamiczne?	316
11.10. Wielometody o stałym czasie rozprowadzania	321
11.11. BasicDispatcher i BasicFastDispatcher w roli wytycznych	324

11.12. Naprzód	325
11.13. Podsumowanie	327
11.14. Podwójne rozprowadzanie — na skróty	328

A

Minimalistyczna biblioteka wielowątkowa	333
A.1. Krytyka wielowątkowości	334
A.2. Podejście à la Loki	335
A.3. Operacje niepodzielne na typach całkowitoliczbowych	335
A.4. Muteksy	337
A.5. Semantyka blokowania w programowaniu obiektowym	339
A.6. Opcjonalny modyfikator volatile	341
A.7. Semafor, zdarzenia i inne	342
A.8. Podsumowanie	342
Bibliografia	343
Skorowidz	345

2

Techniki

Ten rozdział będzie prezentował zbiór technik C++, które będą stosowane w dalszej części książki. Są to techniki pomocne w rozmaitych kontekstach, a jako takie najczęściej bardzo ogólne i użyteczne uniwersalnie, co pozwala znajdować dla nich zastosowania również w innych dziedzinach. Niektóre z tych technik, jak częściowa specjalizacja szablonu, to mechanizmy wbudowane w język; inne, jak asercje statyczne (asercje czasu kompilacji), wymagają dodatkowego kodu obsługującego.

W rozdziale przedstawione zostaną następujące techniki (tudzież narzędzia):

- Asercje statyczne.
- Częściowa specjalizacja szablonu.
- Klasy lokalne.
- Odwzorowania typ-wartość (szablon klasy `Int2Type` i `Type2Type`).
- Szablon klasy `Select` — narzędzie do wyboru typu w czasie kompilacji, zależnie od wartości warunku logicznego.
- Statyczne (w czasie kompilacji) wykrywanie dziedziczenia i możliwości konwersji.
- `TypeInfo` (poręczny dodatek do `std::type_info`).
- `Traits` — kolekcja cech typów stosowalnych do dowolnych typów C++.

Rozpatrywana z osobna, każda z tych technik (oraz kod ją realizujący) sprawia wrażenie trywialnej; zazwyczaj składa się na nią pięć do dziesięciu wierszy łatwego do ogarnięcia kodu. Ale wszystkie one mają istotną cechę: są technikami „nieterminalnymi”, co należy rozumieć tak, że dają się łączyć z innymi technikami, czego wynikiem są idiomy wyższego poziomu. Razem stanowią zaś solidny fundament usług wspomagających budowanie efektywnych struktur architektonicznych projektu.

Omówienia technik nie są nadto suche, bo uzupełniają je przykłady. Zawartość rozdziału polecam również do referencji przy dalszej lekturze książki, gdzie pojawią się odniesienia do poszczególnych technik i ich konglomeratów.

2.1. Asercje statyczne

Od kiedy zaczęto w C++ programować w sposób uogólniony, pojawiła się również potrzeba lepszej statycznej kontroli (i lepszych, konfigurowalnych komunikatów o błędach), to znaczy kontroli realizowanej w czasie kompilacji.

Załóżmy dla przykładu, że pracujemy nad funkcją do bezpiecznego rzutowania typów. Chcemy wykonać rzutowanie jednego typu na inny przy zapewnieniu zachowania kompletu informacji: chcemy zapobiec rzutowaniu typów większych na mniejsze.

```
template <class To, class From>
To safe_reinterpret_cast(From from)
{
    assert(sizeof(From) <= sizeof(To));
    return reinterpret_cast<To>(from);
}
```

Funkcję tę wywołuje się przy użyciu składni identycznej jak dla zwyczajnego rzutowania w C++:

```
int i = ...;
char *p = safe_reinterpret_cast<char*>(i);
```

Argument szablonu `To` określa się jawnie, a typ `From` kompilator wysnuwa sam na podstawie typu argumentu wywołania, czyli na podstawie typu zmiennej `i`. Asercja nałożona na porównanie typów pozwala wymusić, aby typ docelowy pomieścił całość informacji przechowywanej w typie źródłowym. Dzięki temu zabezpieczeniu powyższy kod albo wykona skuteczną konwersję typu¹, albo spowoduje niespełnioną asercję w czasie wykonania.

Z oczywistych względów wolelibyśmy wykrywać tego rodzaju błędy już w czasie kompilacji. Przecież feralne rzutowanie może na przykład zostać użyte w rzadko wykonywanej ścieżce przebiegu programu i niekoniecznie musi się ujawnić w testach. Dalej, przy przenoszeniu aplikacji na nową platformę albo do nowego kompilatora trudno zapamiętać wszystkie potencjalne aspekty nieprzenośności programu; tak czy inaczej, błąd rzutowania może zagnieździć się w kodzie na dłuższy czas i spowodować krach programu dopiero dużo później, najpewniej na oczach klienta.

Jest nadzieja: wyrażenie obliczane w ramach asercji jest stałą czasu kompilacji, co oznacza, że potencjalnie jej kontrolę mógłby wykonać kompilator, a nie środowisko wykonawcze. Pomysł polega na przekazaniu do kompilatora konstrukcji języka C++ tak użytej, aby była dozwolona dla wyrażeń niezerowych i niedozwolona dla wyrażeń, których obliczona statycznie wartość wynosi zero. W ten sposób, jeśli w kodzie pojawi się wyrażenie o wartości zerowej, kompilator zasygnalizuje błąd czasu kompilacji.

Najprostszym mechanizmem asercji statycznych, działającym również w języku C, jest mechanizm oparty na zakazie deklarowania tablicy o zerowej liczbie elementów (Van Horn 1997):

```
#define STATIC_CHECK(expr) { char unnamed[(expr) ? 1 : 0]; }
```

Teraz, jeśli napiszemy:

```
template <class To, class From>
To safe_reinterpret_cast(From from)
```

¹ Poprawną na większości komputerów — z `reinterpret_cast` nigdy nie można mieć absolutnej pewności — *przyp. autora*.

```

{
    STATIC_CHEK(sizeof(From) <= sizeof(To));
    return reinterpret_cast<To>(from);
}
...
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);

```

i jeśli w danym systemie wskaźniki są większe niż znaki, kompilator zaprotestuje przeciwko utworzeniu tablicy o zerowej liczbie elementów, protestując tym samym przeciwko niepożądanemu przez nas konwersji.

Problem w tym, że otrzymany wtedy komunikat o błędzie nie będzie ani słowem wspominał o próbie konwersji; najprawdopodobniej będzie informował o niemożności utworzenia tablicy o zerowym rozmiarze. Nijak z tego nie wynika, że rozmiar typu `char` jest mniejszy od rozmiaru typu wskaźnikowego. Udostępnienie własnego komunikatu o błędzie jest trudne, zwłaszcza jeśli ma to być mechanizm przenośny. Komunikaty o błędach nie podlegają żadnym regułom; steruje nimi wyłącznie kompilator. Na przykład w przypadku niezdefiniowanej zmiennej kompilator nie ma nawet obowiązku podawać nazwy tej zmiennej w komunikacie o błędzie.

Lepszym rozwiązaniem jest sięgnięcie po szablon o odpowiednio sugestywnej nazwie; przy odrobinie szczęścia kompilator wymieni nazwę szablonu w komunikacie o błędzie.

```

template<bool> struct CompileTimeError;
template<> struct CompileTimeError<>true> {};

```

```

#define STATIC_CHECK(expr) \
    (CompileTimeError<expr> != 0>());

```

`CompileTimeError` to szablon z parametrem pozatypowym (parametryzowany stałą typu `bool`). Szablon jest zdefiniowany wyłącznie dla wartości `true` tej stałej. Jeśli spróbujemy skonkretyzować szablon `CompileTimeError` dla wartości `false`, kompilator będzie zmuszony do zgłoszenia błędu z komunikatem o niezdefiniowanej specjalizacji `CompileTimeError<false>`. Taki komunikat jest już dużo lepszą wskazówką co do przyczyny błędu.

Nie jest to, rzecz jasna, rozwiązanie już doskonałe. Co z treścią komunikatu o błędzie? Można by przekazywać do `STATIC_CHECK` dodatkowy parametr i jakoś wymuszać ujawnienie go w komunikacie o błędzie. Sęk w tym, że przekazany komunikat o błędzie musi być dozwolonym identyfikatorem C++ (bez znaków odstępu, niezaczynającym się od cyfry itd.). Ten tok myślenia prowadzi nas do ulepszonej wersji szablonu `CompileTimeError`, widocznej poniżej. W sumie nazwa `CompileTimeError` przestaje wtedy być odpowiednio sugestywna; za chwilę okaże się, że znacznie lepszą nazwą jest `CompileTimeChecker`.

```

template<bool> struct CompileTimeChecker
{
    CompileTimeChecker(...);
};
template<> struct CompileTimeChecker<false> {};
#define STATIC_CHECK(expr, msg) \
{\
    class ERROR_##msg; \
    (void)sizeof(CompileTimeChecker<\
        (expr) != 0>((ERROR_##msg())));\
}

```

Załóżmy, że `sizeof(char) < sizeof(void*)` (standard nie gwarantuje prawdziwości takiej relacji w każdym przypadku). Zobaczmy, co się stanie, kiedy napiszemy:

```
template <class To, class From>
To safe_reinterpret_cast(From from)
{
    STATIC_CHECK(sizeof(From) <= sizeof(To),
        Destination_Type_Too_Narrow);
    return reinterpret_cast<To>(from);
}
...
void* somePointer = ...;
char c = safe_reinterpret_cast<char>(somePointer);
```

Po rozwinięciu makrodefinicji `STATIC_CHECK` kod funkcji `safe_reinterpret_cast` będzie prezentował się następująco:

```
template <class To, class From>
To safe_reinterpret_cast(From from)
{
    {
        class ERROR_Destination_Type_Too_Narrow {};
        (void)sizeof((
            CompileTimeChecker<sizeof(From) <= sizeof(To)>(
                ERROR_Destination_Type_Too_Narrow())));
    }
    return reinterpret_cast<To>(from);
}
```

Powyższy kod definiuje *klasę lokalną* o nazwie `ERROR_Destination_Type_Too_Narrow`, o pustym ciele. Następnie tworzy tymczasową wartość typu `CompileTimeChecker<sizeof(From) <= sizeof(To)>`, inicjalizowaną wartością tymczasową typu `ERROR_Destination_Type_Too_Narrow`. Na koniec operator `sizeof` oblicza rozmiar wynikowej wartości tymczasowej.

Gdzie jest trik? Otóż specjalizacja `CompileTimeChecker<true>` posiada konstruktor akceptujący dowolne argumenty (lista parametrów w postaci wielokropka). Oznacza to, że kiedy sprawdzane wyrażenie jest obliczane jako wartość `true`, wynikowy program jest poprawny. Natomiast kiedy porównanie pomiędzy typami da wartość `false`, dojdzie do błędu kompilacji: kompilator nie będzie mógł znaleźć konwersji z typu `ERROR_Destination_Type_Too_Narrow` na typ `CompileTimeChecker<false>`. A najlepsze, że porządny kompilator wypisze wtedy całkiem dokładny komunikat o błędzie, w rodzaju: „Nie można skonwertować `ERROR_Destination_Type_Too_Narrow` na `CompileTimeChecker<false>`”.

Jesteśmy w domu!

2.2. Częściowa specjalizacja szablonu

Częściowa specjalizacja szablonu pozwala na specjalizowanie szablonu klasy dla podzbioru konkretyzacji możliwych dla tego szablonu.

Przypomnijmy na razie pełną specjalizację szablonu. Otóż posiadając szablon klasy `Widget`:

```
template <class Window, class Controller>
class Widget
{
    ... uogólniona implementacja klasy Widget ...
};
```

możemy jawnie specjalizować szablony klasy `Widget` dla wybranego zestawu typów, na przykład:

```
template <>
class Widget<ModalDialog, MyController>
{
    ... specjalizowana implementacja klasy Widget ...
};
```

gdzie `ModalDialog` i `MyController` to klasy definiowane przez aplikację.

Kiedy kompilator napotyka w kodzie definicję specjalizacji szablonu `Widget`, wykorzystuje tę specjalizowaną implementację wszędzie tam, gdzie definiujemy obiekt typu `Widget<ModalDialog, MyController>`, a dla wszelkich innych konkretyzacji szablonu używa definicji ogólnej `Widget`.

Niekiedy jednak chcemy specjalizować `Widget` dla dowolnych wartości parametru `Window` w połączeniu z typem `MyController`. Potrzebujemy wtedy częściowej specjalizacji szablonu:

```
// częściowa specjalizacja szablonu Widget
template <class Window>
class Widget<Window, MyController>
{
    ... częściowo specjalizowana implementacja klasy Widget ...
};
```

W częściowej specjalizacji szablonu klasy dookreślamy zazwyczaj tylko niektóre z wymaganych parametrów szablonu, a resztę pozostawiamy w ujęciu ogólnym. Przy konkretyzowaniu szablonu klasy w programie kompilator próbuje dopasować najlepszą wersję szablonu. Algorytm dopasowania jest bardzo zawiły i ścisły, co pozwala na dość nietypowe zastosowania częściowych specjalizacji. Na przykład założymy, że posiadamy szablony klasy `Button` parametryzowany jednym parametrem. Wtedy, nawet jeśli wyspecjalizowaliśmy już `Widget` dla dowolnego parametru typowego `Window` i klasy `MyController`, możemy dalej specjalizować szablony `Widget` dla wszystkich konkretyzacji szablonu `Button` w połączeniu z typem `MyController`:

```
template <class ButtonArg>
class Widget<Button<ButtonArg>, MyController>
{
    ... dalsza specjalizacja szablonu klasy Widget ...
};
```

Jak widać, możliwości częściowej specjalizacji szablonów są dość niezwykłe. Przy konkretyzowaniu szablonu kompilator przeprowadza dopasowanie wzorca istniejących częściowych i pełnych specjalizacji w poszukiwaniu najlepszego dopasowania; w ten sposób zyskujemy niebagatelnie elastyczność.

Niestety, częściowa specjalizacja szablonów nie dotyczy funkcji — czy to samodzielnych, czy metod szablonów klas — co poniekąd redukuje elastyczność i precyzję specjalizacji:

- Chociaż można *całkowicie specjalizować* metody szablonu klasy, nie można *częściowo specjalizować* metod.
- Nie można częściowo specjalizować szablonów funkcji o zasięgu przestrzeni nazw. Najbliższe modelowi częściowej specjalizacji funkcji jest przeciążanie. W ujęciu praktycznym oznacza to, że precyzyjna specjalizacja dotyczy jedynie parametrów wywołania funkcji, ale nie może już dotyczyć wartości zwracanej ani typów wykorzystywanych wewnętrznie. Na przykład:

```
template <class T, class U> T Fun(U obj); //szablon główny
template <class U> void Fun<void, U>(U obj); //niedozwolona specjalizacja częściowa
template <class T> T Fun(Window obj); //dozwolona specjalizacja (przeciążenie)
```

Brak możliwości precyzyjnego częściowego specjalizowania funkcji ułatwia pracę programistom kompilatorów, ale dla programistów aplikacji ma same słabe strony. Niektóre z narzędzi prezentowanych w dalszym omówieniu (na przykład `Int2Type` i `Type2Type`) powstały wyłącznie jako odpowiedź na tę niedogodność.

W niniejszej książce częściowe specjalizacje szablonów wykorzystywane są bardzo szeroko. Chociażby cały rozdział 3. omawiający mechanizm list typów jest zbudowany w oparciu właśnie o specjalizacje częściowe.

2.3. Klasy lokalne

Klasy lokalne są ciekawe jako mało znana cecha języka C++. Otóż klasę można zdefiniować wewnątrz funkcji, jak poniżej:

```
void Fun()
{
    class Local
    {
        ... składowe danych...
        ... definicje metod ...
    };
    ... kod odwołujący się do klasy Local ...
}
```

Są pewne ograniczenia: klasy lokalne nie mogą definiować składowych statycznych i nie mają dostępu do niestycznych zmiennych lokalnych funkcji. Klasy lokalne są interesujące głównie z tego powodu, że można ich używać w funkcjach szablonowych. Klasy lokalne definiowane we wnętrzu szablonu funkcji mogą korzystać z parametrów szablonu funkcji.

Poniższy szablon funkcji `MakeAdapter` przystosowuje jeden interfejs do wymogów innego interfejsu. `MakeAdapter` implementuje interfejs w locie, za pomocą klasy lokalnej. Klasa lokalna przechowuje składowe uogólnionych typów.

```
class Interface
{
public:
    virtual void Fun() = 0;
    ...
};
```

```

template <class T, class P>
Interface MakeAdapter(const T& obj, const P& arg)
{
    class Local : public Interface
    {
    public:
        Local(const T& obj, const P& arg)
        : obj_(obj), arg_(arg) {}
        virtual void Fun()
        {
            obj_.Call(arg_);
        }
    private:
        T obj_;
        P arg_;
    };
    return new Local(obj, arg);
}

```

Można łatwo udowodnić, że każdy idiom używający klasy lokalnej można zaimplementować z użyciem szablonu klasy poza funkcją. Innymi słowy, klasy lokalne nie są mechanizmem umożliwiającym konstrukcje unikalne. Z drugiej strony, klasy lokalne mogą uprościć implementowanie i poprawiają lokalizację (ograniczenie zasięgu) symboli.

Klasy lokalne posiadają jednak pewną cechę unikatową: są klasami *finalnymi*. Użytkownicy zewnętrzni nie mogą wyprowadzać pochodnych klasy ukrytej wewnątrz funkcji. Bez klas lokalnych podobny efekt wymagałby dodania nienazwanej przestrzeni nazw w osobnej jednostce kompilacji.

Klasy lokalne wykorzystamy w rozdziale 11. do utworzenia funkcji trampolinowych.

2.4. Mapowanie stałych na typy

Prosty szablon opisany pierwotnie w (Alexandrescu 2000b) może być wielce pomocny przy wielu idiomach programowania uogólnionego. Oto on:

```

template <int v>
struct Int2Type
{
    enum { value = v };
};

```

Szablon `Int2Type` definiuje odrębny typ dla każdej przekazanej odrębnej stałej wartości całkowitej. To dlatego, że odrębne konkretyzacje szablonów stanowią pełnoprawne odrębne typy; z tego względu `Int2Type<0>` to typ odmienny niż `Int2Type<1>` itd. Dodatkowo wartość generująca typ jest „utrwalana” w składowej wyliczeniowej `value`.

Szablonu `Int2Type` można używać wszędzie tam, gdzie chcemy szybko „otypować” stałe wartości całkowite. W ten sposób można zrealizować rozproszanie wywołań do różnych funkcji, zależnie od wyniku wyrażenia obliczanego w czasie kompilacji. Efektywnie można w ten sposób zrealizować statyczne rozproszanie wywołań na bazie stałych wartości całkowitych.

Szablon w rodzaju `Int2Type` jest zazwyczaj stosowany, kiedy spełnione są poniższe warunki:

- Zachodzi potrzeba wywołania jednej z wielu wersji funkcji, zależnie od stałej znanej w czasie kompilacji.
- Zachodzi potrzeba rozprawadzenia tego wywołania w czasie kompilacji.

W przypadku rozprawadzenia w czasie wykonania można uciec się do prostej konstrukcji kaskadowych instrukcji warunkowych `if-else` albo instrukcji wyboru `switch`. Niekiedy jednak jest to niepożądane. Instrukcje `if-else` wymagają poprawnego skompilowania obu alternatywnych gałęzi kodu, nawet jeśli wartość warunku znana jest już w czasie kompilacji. Niejasne? Za chwilę się wyjaśni.

Weźmy taki przypadek: zaprojektowaliśmy uogólniony kontener `NiftyContainer`, parametryzowany typem przechowywanych obiektów:

```
template <class T> class NiftyContainer
{
    ...
};
```

Powiedzmy, że `NiftyContainer` przechowuje wskaźniki do obiektów typu `T`. Aby powielić obiekt zawarty w `NiftyContainer`, trzeba albo wywołać jego konstruktor kopiujący (dla typów niepolimorficznych), albo wywołać metodę wirtualną `Clone()` (dla typów polimorficznych). Informację o trybie powielania uzyskujemy od użytkownika za pośrednictwem parametru szablonu w postaci stałej logicznej:

```
template <typename T, bool IsPolymorphic>
class NiftyContainer
{
    ...
    void DoSomething()
    {
        T* pSomeObj = ...;
        if (isPolymorphic)
        {
            T* pNewObj = pSomeObj->Clone();
            ... algorytm dla typów polimorficznych ...
        }
        else
        {
            T* pNewObj = new T(*pSomeObj);
            ... algorytm dla typów monomorficznych ...
        }
    }
};
```

Sęk w tym, że kompilator nie przepuści takiego kodu. Ponieważ w wersji polimorficznej algorytm wykorzystuje metodę `pObj->Clone()`, wywołanie `NiftyContainer::DoSomething()` nie skompiluje się dla żadnego typu, który nie definiuje metody `Clone()`. Prawda, że w czasie kompilacji jest tu oczywiste, którą gałąź instrukcji warunkowej program faktycznie wykona, ale dla kompilatora nie jest to argumentem — również kod w nieużywanej gałęzi musi być poprawny, choćby miał być później wytrzebiony z programu w ramach optymalizacji i usuwania „martwego” kodu.

Próba wywołania `DoSomething` dla typu `NiftyContainer<int,false>` nieodwołalnie zatrzyma kompilację na wierszu wywołania `pObj->Clone()`.

Błąd kompilacji może pojawić się zresztą również w drugiej gałęzi kodu, dla wersji niepolimorficznej, w której następuje próba utworzenia obiektu za pomocą wywołania `new T(*pObj)`. Błąd ten wystąpi, kiedy typ `T` nie będzie udostępniał konstruktora kopiującego (na przykład przez oznaczenie konstruktora jako prywatnego) — co zresztą w przypadku typów polimorficznych jest zalecane.

Byłoby znacznie lepiej, gdyby kompilator nie próbował nawet kompilować kodu, który i tak się nie wykona; jak to osiągnąć?

Okazuje się, że istnieje kilka rozwiązań tego problemu, a wśród nich `Int2Type` jest jednym z bardziej eleganckich. Transformuje on ad hoc wartość typu logicznego `IsPolymorphic` na dwa różne typy, reprezentujące wartości `true` i `false`. Wystarczy użyć `Int2Type<IsPolymorphic>` z prostym przeciążaniem:

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
private:
void DoSomething(T* pObj, Int2Type<true>)
{
    T* pNewObj = pObj->Clone();
    ... algorytm polimorficzny ...
}
void DoSomething(T* pObj, Int2Type<false>)
{
    T* pNewObj = new T(*pObj);
    ... algorytm niepolimorficzny ...
}
public:
void DoSomething(T* pObj)
{
    DoSomething(pObj, Int2Type<isPolymorphic>());
}
};
```

`Int2Type` jako mechanizm konwersji wartości na typ jest bardzo poręczny. Wystarczy przekazać wartość tymczasową uzyskanego typu do przeciążonej funkcji. Przeciążenie wybiera wtedy pożądaną wariant algorytmu.

Sztuczka działa, ponieważ kompilator nie kompiluje funkcji szablonowych, które nie są używane — sprawdza jedynie ich poprawność składniową. A w kodzie szablonowym rozprawdanie wywołań zazwyczaj odbywa się statycznie (w czasie kompilacji).

Narzędzie `Int2Type` można obejrzeć w działaniu w kilku miejscach w bibliotece `Loki`, a także w rozdziale 11. przy omawianiu wielometod. Mamy tam szablon klasy w roli mechanizmu podwójnego rozprawdania, z parametrem typu logicznego określającym opcję rozprawdania symetrycznego.

2.5. Odwzorowanie typu na typ

W podrozdziale 2.2 padło stwierdzenie, że nie można częściowo specjalizować funkcji szablonowych. Niekiedy jednak zachodzi potrzeba zasymulowania takiej możliwości. Weźmy poniższą funkcję:

```
template <class T, class U>
T* Create(const U& arg)
{
    return new T(arg);
}
```

Metoda `Create` tworzy nowy obiekt, przekazując argument wywołania do konstruktora obiektu.

Załóżmy teraz, że w aplikacji przyjęliśmy regułę: obiekty typu `Widget` są nieetykalne jako kod zastany (nie mamy wpływu na ich implementację), a ich konstruktor przyjmuje dwa argumenty wywołania, z których drugi jest stałą wartością, na przykład `-1`. Z kolei nasze własne klasy, wprowadzone z typu `Widget`, są pozbawione tego udziwnienia.

Jak można wyspecjalizować metodę `Create` tak, żeby traktowała klasę `Widget` inaczej niż wszystkie inne typy? Oczywiście rozwiązaniem byłoby utworzenie osobnej funkcji `CreateWidget`, specjalnie dla tego przypadku szczególnego. Niestety, w ten sposób pozbedziemy się jednolitego interfejsu tworzenia obiektów typu `Widget` i obiektów pochodnych względem tego typu. To z kolei niewieczy użyteczność metody `Create` w kodzie uogólnionym.

Nie możemy częściowo specjalizować funkcji, to znaczy nie wolno nam napisać czegoś takiego:

```
// niedozwolony kod - nie próbujcie tego w domu
template <class U>
Widget* Create<Widget, U>(const U& arg)
{
    return new Widget(arg, -1);
}
```

Pod nieobecność częściowego specjalizowania funkcji uciekamy się do jedyne go dostępnego narzędzia: przeciążania. Rozwiązaniem byłoby przekazanie do `Create` atrapowego obiektu typu `T` i odwołanie się do przeciążania:

```
template <class T, class U>
T* Create(const U& arg, T /* atrapa */)
{
    return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Widget /* atrapa */)
{
    return new Widget(arg, -1);
}
```

Rozwiązanie to wprowadza narzut związany z tworzeniem obiektów (o nieokreślonej złożoności), które pozostają nieużywane. Potrzebujemy więc możliwie skromnego środka do przeniesienia informacji o typie `T` do metody `Create`. Tu do akcji wkroczy `Type2Type`: jest to reprezentant typu, jego tani identyfikator, który można przekazać do przeciążonej funkcji.

Oto definicja szablonu `Type2Type`:

```
template <typename T>
struct Type2Type
{
    typedef T OriginalType;
};
```

`Type2Type` to klasa pozbawiona jakiegokolwiek „fizycznej” składowej, ale parametryzowana różnymi typami daje różne konkretyzacje — dokładnie tego nam potrzeba.

Teraz możemy napisać tak:

```
// implementacja metody Create na bazie przeciążania i Type2Type
template <class T, class U>
T* Create(const U& arg, Type2Type<T>)
{
    return new T(arg);
}
template <class U>
Widget* Create(const U& arg, Type2Type<Widget>)
{
    return new Widget(arg, -1);
}
// użycie metody Create()
String* pStr = Create("Ahoj", Type2Type<String>());
Widget* pW = Create(100, Type2Type<Widget>());
```

Drugi parametr wywołania `Create` służy jedynie do wybrania odpowiedniego przeciążenia. Teraz możemy specjalizować `Create` dla różnych konkretyzacji `Type2Type`, reprezentujących różne typy występujące w aplikacji (i o różnych wymaganiach względem składni kreacji obiektów).

2.6. Wybór typu

Niekiedy kod uogólniony musi wybrać któryś z typów, zależnie od wartości stałej typu logicznego.

Zalóżmy, że w kontenerze `NiftyContainer`, omawianym w podrozdziale 2.4, chcemy w roli magazynu obiektów użyć kontenera `std::vector`. Rzecz jasna, typów polimorficznych nie można przechowywać przez wartość, więc w magazynie umieścimy wskaźniki. Z drugiej strony, typy monomorficzne jak najbardziej nadają się do przechowywania przez wartość i niekiedy może to być bardziej efektywne.

W przykładowym szablonie klasy:

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
};
```

chcemy przechowywać albo `vector<T*>` (kiedy `IsPolymorphic` ma wartość `true`), albo `vector<T>` (dla `IsPolymorphic` równego `false`). Zasadniczo potrzebujemy więc definicji typu, na przykład `ValueType`, która w zależności od wartości `IsPolymorphic` będzie reprezentowała `T*` albo `T`.

Można w tym celu użyć szablonu cechy (ang. *traits*) (Alexandrescu 2000a), jak tutaj:

```
template <typename T, bool isPolymorphic>
struct NiftyContainerValueTraits
{
    typedef T* ValueType;
};
template <typename T>
struct NiftyContainerValueTraits<T, false>
{
    typedef T ValueType;
};
template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    typedef NiftyContainerValueTraits<T, isPolymorphic>
        Traits;
    typedef typename Traits::ValueType ValueType;
};
```

Ten sposób jest jednak niepotrzebnie zawily. Co więcej, niespecjalnie dobrze się skaluje: dla każdego wyboru typu potrzebujemy nowego, osobnego szablonu cechy.

Szablon klasy `Select` udostępniony w bibliotece `Loki` oferuje znacznie prostszy mechanizm wyboru typów. Jego definicja wykorzystuje częściową specjalizację szablonu:

```
template <bool flag, typename T, typename U>
struct Select
{
    typedef T Result;
};
template <typename T, typename U>
struct Select<false, T, U>
{
    typedef U Result;
};
```

Sposób działania tej specjalizacji można opisać tak: jeśli flag ma wartość `true`, kompilator używa pierwszej (uogólnionej) definicji, więc `Result` będzie się równać `T`. Z kolei jeśli flag ma wartość `false`, do gry wkracza specjalizacja szablonu i definicja `Result` jest rozwijana jako `U`.

Z takim oprzyrządowaniem definicja `NiftyContainer::ValueType` jest znacznie łatwiejsza:

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    typedef typename Select<isPolymorphic, T*, T>::Result
```

```

    ValueType;
    ...
};

```

2.7. Statyczne wykrywanie dziedziczenia i możliwości konwersji

Przy implementowaniu funkcji i klas szablonowych niejednokrotnie powstaje wątpliwość: czy dla dwóch arbitralnych typów T i U , o których twórcy szablonu nic nie wiadomo, można wykryć relację dziedziczenia pomiędzy U i T ? Wykrycie takiej relacji w czasie kompilacji to klucz do implementowania zaawansowanych optymalizacji w bibliotekach uogólnionych. W uogólnionej funkcji można na przykład wykorzystać zoptymalizowany algorytm, o ile używana w niej klasa implementuje pewien określony interfejs. Wykrycie obecności tego interfejsu w czasie kompilacji oznacza uniknięcie rzutowania `dynamic_cast`, które w czasie wykonania jest dość kosztowne.

Wykrywanie dziedziczenia odbywa się na bazie bardziej ogólnego mechanizmu, to znaczy wykrywania możliwości konwersji. Ów ogólniejszy problem można sformułować tak: jak wykryć, czy dowolny typ T obsługuje automatyczną konwersję na dowolny typ U ?

Istnieje rozwiązanie tego problemu oparte na operatorze `sizeof`. Użyteczność `sizeof` jest doprawdy zadziwiająca: można go zastosować wobec dowolnie złożonego wyrażenia, a operator zwróci rozmiar tego wyrażenia bez obliczania go w czasie wykonania — wszystko w czasie kompilacji! Oznacza to, że operator `sizeof` uwzględnia przeciążanie, konkretyzację szablonów, reguły konwersji — wszystko, co może uczestniczyć w poprawnym wyrażeniu języka C++. W rzeczy samej `sizeof` zawiera w sobie kompletne oprzyrządowanie do dedukcji typu wyrażenia; ostatecznie przecież `sizeof` ignoruje wartość wyrażenia, a zwrócony rozmiar to rozmiar typu wyrażenia².

Pomysł na wykrywanie możliwości konwersji polega na użyciu operatora `sizeof` w połączeniu z funkcjami przeciążonymi. Udostępnimy dwa przeciążenia funkcji: jedno przyjmujące typ docelowy konwersji (U) i drugie przyjmujące argument dowolnego innego typu. Możemy wtedy wywołać funkcję przeciążoną z argumentem w postaci tymczasowej wartości typu T , którego zgodność z U chcemy sprawdzić. Jeśli w ramach rozstrzygnięcia przeciążenia dojdzie do wywołania funkcji przyjmującej argument typu U , wiadomo, że typ T jest zgodny z typem U ; jeśli wywołane zostanie drugie przeciążenie, wiadomo, że T nie da się skonwertować na U . Aby wykryć funkcję wywołaną w ramach tego sprawdzianu, zaaranżujemy dwa przeciążenia tak, aby ich typy zwracane były różnych rozmiarów — wnioskowanie można będzie wtedy oprzeć się na operatorze `sizeof` odniesionym do wartości zwracanej przeciążenia. Dokładne typy przeciążenia są nieistotne, byleby ich rozmiar był różny.

Utwórzmy najpierw dwa typy o różnych rozmiarach (to nie takie oczywiste, bo na przykład choć przeważnie `char` i `long double` są różnych rozmiarów, to standard tego nie gwarantuje). Najprostsze byłoby coś takiego:

```

typedef char Small;
class Big { char dummy[2]; };

```

² Istnieje propozycja uzupełnienia C++ o operator `typeof`, to znaczy operator zwracający typ wyrażenia. Obecność operatora `typeof` ogromnie uprościłaby dużą część kodu szablonowego, czyniąc go bardziej zrozumiałym. GNU C++ implementuje już taki operator w ramach własnego rozszerzenia języka. Jasne jest, że `typeof` dzieliłby z operatorem `sizeof` znaczną część jego wewnętrznej implementacji — `sizeof` przecież już teraz musi jakoś określać typ wyrażenia — *przyp. autora*.

Z definicji `sizeof(Small)` wynosi 1. Rozmiar typu `Big` nie jest dokładnie znany, ale na pewno będzie większy od 1 — to wszystko, czego nam trzeba do rozróżnienia typów.

Teraz para przeciążeń. Jedno z nich ma przyjmować `U` i zwracać, powiedzmy, obiekt typu `Small`:

```
Small Test(U);
```

Ale jak napisać drugą funkcję, która przyjmuje „dowolny inny typ”? Szablon nie jest rozwiązaniem, ponieważ przy rozstrzygnięciu przeciążenia szablon zawsze będzie kwalifikowany jako „najlepsze dopasowanie”, co zakryje konwersję. Potrzebujemy czegoś, co jest „gorsze” (w sensie jakości dopasowania) od konwersji automatycznej — to znaczy potrzeba nam takiej konwersji, która jest uruchamiana jedynie przy braku konwersji automatycznej. Szybki przegląd reguł konwersji typów argumentów wywołania funkcji pozwoli wytypować wielokropek, który jest dopasowaniem najgorszym z możliwych — znajduje się na samym końcu listy rozpatrywanych konwersji. Dokładnie tego szukaliśmy:

```
Big Test(...);
```

(Przekazywanie obiektu `C++` do funkcji z wielokropkiem prowokuje niezdefiniowany wynik, ale to nie jest istotne; nie będziemy faktycznie wywoływać funkcji; ba, funkcja nie jest nawet zaimplementowana — pamiętajmy, że `sizeof` nie oblicza wartości wyrażenia, jedynie jego typ).

Teraz zastosujemy operator `sizeof` do próbnego wywołania funkcji `Test`, przekazując do niej obiekt typu `T`:

```
const bool convExists = sizeof(Test(T())) == sizeof(Small);
```

I już! Funkcja `Test` otrzymuje w wywołaniu obiekt skonstruowany domyślnie (`T()`), a następnie operator `sizeof()` ustala rozmiar wyniku takiego wyrażenia. Wynik może mieć albo rozmiar `sizeof(Small)`, albo `sizeof(Big)`, zależnie od tego, czy typ `T` posiada możliwość automatycznej konwersji na typ `U`, czy nie.

Został tylko jeden problem: otóż jeśli `T` posiada prywatny konstruktor domyślny, nie uda się skompilować wyrażenia `T()`, a więc i całego naszego sprawdzianu konwersji. Na szczęście to również można obejść — wystarczy użyć funkcji-atrapy z typem wartości zwracanej `T` (pamiętajmy, wciąż operujemy w kontekście operatora `sizeof`, to znaczy bez faktycznego obliczania wartości wyrażenia — a więc i bez wywoływania funkcji, tworzenia obiektów itd.) Kompilator nie będzie miał wtedy powodów do narzekania:

```
T MakeT(); // bez implementacji
const bool convExists = sizeof(Test(MakeT())) == sizeof(Small);
```

Nawiasem mówiąc, czy to nie cudowne, że tyle można zdziałać za pomocą prostych funkcji, jak `MakeT` czy `Test`, które nie tylko nic nie robią, ale wręcz nie istnieją?

Skoro mamy gotowy mechanizm, upakujmy całość w szablon klasy, który ukryje wszystkie szczegóły wnioskowania o typach i udostępni jedynie wynik sprawdzianu możliwości konwersji.

```
template <class T, class U>
class Conversion
{
    typedef char Small;
    class Big { char dummy[2]; };
    static Small Test( const U& );
};
```

```

    static Big Test(...);
    static T MakeT();
public:
    enum { exists =
sizeof(Test(MakeT())) == sizeof(Small) };
};

```

Klasę sprawdzianu konwersji można już przetestować:

```

int main()
{
    using namespace std;
    cout
    << Conversion<double, int>::exists << ' '
    << Conversion<char, char*>::exists << ' '
    << Conversion<size_t, vector<int> >::exists << ' ';
}

```

Ten krótki program wypisuje 1 0 0. Zauważmy, że chociaż `std::vector` implementuje konstruktor przyjmujący argument typu `size_t`, to test konwersji prawidłowo zwraca 0, ponieważ konstruktor ten jest wyłącznie jawny.

W szablonie `Conversion` możemy zaimplementować jeszcze jedną stałą: `sameType`, która będzie miała wartość `true`, kiedy `T` i `U` reprezentują ten sam typ:

```

template <class T, class U>
class Conversion
{
    ... jak powyżej ...
    enum { sameType = false };
};

```

Implementacja `sameType` będzie oparta na częściowej specjalizacji szablonu `Conversion`:

```

template <class T>
class Conversion<T, T>
{
public:
    enum { exists = 1, sameType = 1 };
};

```

Jesteśmy w domu. Za pomocą szablonu `Conversion` możemy teraz bardzo łatwo wykrywać relację dziedziczenia pomiędzy typami:

```

#define SUPERSUBCLASS(T, U) \
    (Conversion<const U*, const T*>::exists && \
    !Conversion<const T*, const void*>::sameType)

```

Jeśli `U` dziedziczy publicznie po typie `T`, ewentualnie jeśli `T` i `U` są w istocie tymi samymi typami, makrodefinicja `SUPERSUBCLASS(T, U)` jest rozwijana do wartości `true`. Makrodefinicja opiera się na próbie ustalenia możliwości konwersji z `const U*` na `const T*`. Taka konwersja dla arbitralnych typów `U` i `T` jest możliwa tylko w trzech przypadkach:

- (1) T jest tego samego typu co U.
- (2) T jest publiczną klasą bazową U.
- (3) T jest typu void.

Ostatni z tych przypadków jest eliminowany przez drugi test w wyrażeniu. W praktyce pierwszy przypadek (T jest tego samego typu co U) akceptujemy jako zdegenerowaną relację dziedziczenia, ponieważ dla celów praktycznych możemy przecież uznać, że każda klasa jest w jakimś sensie swoją własną klasą bazową. Tam, gdzie potrzebny jest silniejszy sprawdzian dziedziczenia, można go zdefiniować następująco:

```
#define SUPERSUBCLASS_STRICT(T, U) \
(SUPERSUBCLASS(T, U) && \
 !Conversion<const T*, const U*>::sameType)
```

Po co dodaliśmy w kodzie modyfikatory const? Otóż chcemy zapobiec nieudanym sprawdzianom w zawsze problematycznych przypadkach typów const. W szablonie dodanie drugiego const do typu już opatrzonego modyfikatorem const jest i tak ignorowane. W skrócie, umieszczając w makrodefinicji SUPERSUBCLASS modyfikator const, ustawiamy się zawsze po bezpiecznej stronie, niczego nie ryzykując.

Dlaczego SUPERSUBCLASS, a nie BASE_OF albo po prostu INHERITS? Z bardzo praktycznego powodu. Pierwotnie w bibliotece Loki stosowana była makrodefinicja INHERITS, ale zapis INHERITS(T, U) zawsze prowokował pytania o kierunek sprawdzianu — sprawdzamy, czy T dziedziczy po U, czy odwrotnie? Zapis SUPERSUBCLASS (klasa bazowa-klasa pochodna) mówi znacznie więcej o tym, jak interpretowaliśmy argumenty makrodefinicji.

2.8. TypeInfo

Standard języka C++ udostępnia klasę `std::type_info`, która daje możliwość analizowania typu obiektów w czasie wykonania. Klasę tę wykorzystuje się zazwyczaj w połączeniu z operatorem `typeid`. Operator `typeid` zwraca referencję do obiektu `type_info` odpowiedniego dla operandu:

```
void Fun(Base* pObj)
{
    // porównanie dwóch obiektów type_info odpowiadających
    // faktycznym typom *pObj i Derived
    if (typeid(*pObj) == typeid(Derived))
    {
        ... hm, pObj wskazuje tak naprawdę obiekt klasy Derived ...
    }
    ...
}
```

Klasa `type_info` oprócz operatorów porównania `operator==` i `operator!=` udostępnia jeszcze dwie metody:

- Metodę `name`, zwracającą tekstową reprezentację typu w postaci ciągu znaków `const char*`. Nie ma gwarancji co do sposobu odwzorowania nazw klas na ciągi znaków, więc nie należy oczekiwać, że `typeid(Widget).name()` każdorazowo zwróci ciąg `"Widget"`. Zgodna ze standar-

dem (choć raczej nie wzorcowa) byłyby więc nawet taka implementacja, która dla każdego typu zwraca ciąg pusty.

- Metodę `before`, wprowadzającą relację porządkowania dla obiektów `type_info`. Za pomocą `type_info::before` można przeprowadzić indeksowanie po obiektach `type_info`.

Niestety, całkiem użyteczne właściwości klasy `type_info` są opakowane w sposób niepotrzebnie utrudniający ich użycie. Klasa `type_info` blokuje konstruktor kopiujący i operator przypisania, co uniemożliwia przechowywanie obiektów `type_info` — możemy jedynie przechowywać wskaźniki. Obiekty zwracane przez `typeid` mają przydział statyczny, więc nie trzeba się martwić o kontrolę zasięgu i czas życia. Trzeba za to się troszczyć o *jednoznaczność wskaźników*.

Standard nie gwarantuje, że każde wywołanie na przykład `typeid(int)` zwróci referencję do tego samego egzemplarza `type_info`. Nie można więc bezpośrednio porównywać wskaźników do obiektów `type_info`. Należałoby raczej przechowywać wskaźniki obiektów `type_info` i następnie porównywać je za pośrednictwem operatora `type_info::operator==` z wyłuskanyimi wskaźnikami.

Gdybyśmy chcieli posortować obiekty `type_info`, znów powinniśmy przechować wskaźniki do `type_info`, a także użyć ich metod `before`. W efekcie, chcąc użyć obiektów `type_info` w porządkujących kontenerach biblioteki STL, będziemy zmuszeni do napisania prostego funktora operującego na wskaźnikach.

Wszystko to jest na tyle uciążliwe, że warto napisać klasę ujmującą `type_info` i udostępniającą wskaźnik do `type_info`, a także posiadającą:

- Komplet metod klasy `type_info`.
- Semantykę wartości (a więc publiczny konstruktor kopiujący i publiczny operator przypisania).
- Proste porównania na bazie operatorów `operator<` i `operator==`.

Loki definiuje taką otoczkę w postaci klasy `TypeInfo`. Klasa ta prezentuje się następująco:

```
class TypeInfo
{
public:
    // konstruktory/destruktory
    TypeInfo(); // potrzebny dla kontenerów
    TypeInfo(const std::type_info&);
    TypeInfo(const TypeInfo&);
    TypeInfo& operator=(const TypeInfo&);
    // metody zgodności
    bool before(const TypeInfo&) const;
    const char* name() const;
private:
    const std::type_info* pInfo_;
};
// operatory porównań
bool operator==(const TypeInfo&, const TypeInfo&);
bool operator!=(const TypeInfo&, const TypeInfo&);
bool operator<(const TypeInfo&, const TypeInfo&);
bool operator<=(const TypeInfo&, const TypeInfo&);
bool operator>(const TypeInfo&, const TypeInfo&);
bool operator>=(const TypeInfo&, const TypeInfo&);
```


Obecność konstruktora konwertującego przyjmującego argument typu `std::type_info` umożliwia bezpośrednio porównywanie obiektów `TypeInfo` z obiektami `std::type_info`:

```
void Fun(Base* pObj)
{
    TypeInfo info = typeid(Derived);
    ...
    if (typeid(*pObj) == info)
    {
        ... hm, pObj wskazuje tak naprawdę obiekt klasy Derived ...
    }
    ...
}
```

Możliwość kopiowania i porównywania obiektów `TypeInfo` jest istotna w wielu przypadkach. Przykłady skutecznego użycia tych obiektów znajdziemy w rozdziale 8. (w wytwórni klonów) oraz w rozdziale 11. (w mechanizmie podwójnego rozprowadzania).

2.9. NullType i EmptyType

Biblioteka Loki definiuje dwa bardzo proste typy: `NullType` i `EmptyType`. Można je wykorzystywać w obliczeniach na typach w celu wyróżnienia przypadków granicznych.

`NullType` jest klasą udostępniającą typom znacznik braku typu:

```
class NullType {};
```

Nikt raczej nie będzie tworzył obiektów tej klasy — jej zadaniem jest jedynie sygnalizowanie: „Nie jestem interesującym typem”. W podrozdziale 2.10 użyjemy `NullType` dla przypadków, w których potrzebujemy składniowej obecności typu, ale ten typ nie ma sensu semantycznego (za pomocą takiego typu można na przykład odpowiadać na pytanie: „Do jakiego typu wskazuje `int`?”). `NullType` będzie też wykorzystywany w listach typów z rozdziału 3. — do oznaczania końca listy i do sygnalizowania „braku typu”.

Drugi z przytoczonych tu pomocniczych typów to `EmptyType`. Łatwo się domyślić, jak wygląda jego definicja:

```
struct EmptyType {};
```

Po typie `EmptyType` można dziedziczyć, można też przekazywać wartości typu `EmptyType`³. Przydaje się on jako domyślny typ w szablonach — w taki sposób jest używany w liście typów z rozdziału 3.

³ Wszystkie składowe `EmptyType` są publiczne, a ponieważ `EmptyType` nie definiuje własnego konstruktora, otrzymuje zestaw konstruktorów domyślnych, w tym konstruktor kopiujący — *przyp. tłum.*

2.10. Cechy typów

Cechy to uogólniona technika programistyczna, pozwalająca na statyczne (realizowane w czasie kompilacji) podejmowanie decyzji na bazie typów — w sposób analogiczny do podejmowania decyzji na bazie wartości, już w czasie wykonania (Alexandrescu 2000a). Dodając taką anegdotyczną już „dodatkową warstwę pośrednią”, możemy rozwiązać szereg problemów inżynierskich — poprzez przeniesienie decyzji związanych z typami poza bezpośredni kontekst podejmowania tych decyzji. W ten sposób kod zyskuje na przejrzystości, jest bardziej czytelny i łatwiejszy do utrzymania.

Zazwyczaj programista będzie pisał własne szablony i klasy cech, odpowiednio do potrzeb. Ale można wyróżnić zestaw cech odnoszących się do każdego, dowolnego typu. Taki zestaw mógłby uprościć programowanie uogólnione, pozwalając na lepsze dopasowanie kodu szablonu do możliwości typu.

Załóżmy dla przykładu, że implementujemy algorytm kopiowania:

```
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result)
{
    for (; first != last; ++first, ++result)
        *result = *first;
    return result;
}
```

Zasadniczo implementacja takiego algorytmu jest zbędna, bo powieliła on algorytm biblioteki standardowej `std::copy`. Ale być może chcemy wyspecjalizować tę operację dla podzbioru szczególnych typów.

Załóżmy, że pracujemy nad kodem dla maszyny wieloprocesorowej, dla której zdefiniowano bardzo szybką funkcję `BitBlast`, i chcemy tę superszybką funkcję wykorzystać dla możliwie dużej liczby przypadków:

```
//prototyp BitBlast w "SIMD_Fundamentals.h"
void BitBlast(const void* src, void* dest, size_t bytes);
```

Funkcja `BitBlast`, jako wybitnie niskopoziomowa, działa wyłącznie na typach elementarnych i prostych strukturach danych. Nie można użyć `BitBlast` z typami o nietrywialnych konstruktorach kopiujących. Chcielibyśmy więc zaimplementować funkcję `Copy` tak, aby wszędzie, gdzie to możliwe, używała szybkiej funkcji `BitBlast`, a dla typów bardziej zaawansowanych stosowała klasyczne kopiowanie iteracyjne, obiekt po obiekcie. Dzięki temu operacja `Copy` na pewnym zbiorze typów będzie „automagicznie” optymalizowana.

Aby to osiągnąć, potrzebujemy dwóch sprawdzianów:

- Czy `InIt` i `OutIt` to zwyczajne wskaźniki (w odróżnieniu od typów złożonych iteratorów)?
- Czy typ wskazywany przez `InIt` i `OutIt` to obiekty dające się kopiować bajtowo?

Jeśli uda się udzielić odpowiedzi na te pytania w czasie kompilacji i jeśli na oba odpowiedzi będzie brzmiała „tak”, to do kopiowania kolekcji można użyć funkcji `BitBlast`. W innym przypadku trzeba zostać przy tradycyjnej implementacji kopiowania.

W rozwiązaniu tego problemu pomocne są cechy typów. Cechy opisywane w tym podrozdziale zawdzięczają bardzo wiele implementacji cech typów zrealizowanej w bibliotece `Boost C++` (`Boost`).

2.10.1. Implementowanie cech wskaźników

Biblioteka Loki definiuje szablon klasy `TypeTraits`, ujmujący zestaw ogólnych cech typów. Szablon `TypeTraits` wykorzystuje wewnętrznie specjalizację szablonu i udostępnia wyniki specjalizacji.

Implementacja większości cech typów sprowadza się do specjalizacji pełnej albo częściowej (patrz podrozdział 2.2). Dla przykładu poniższy kod określa, czy `T` jest wskaźnikiem:

```
template <typename T>
class TypeTraits
{
private:
    template <class U> struct PointerTraits
    {
        enum { result = false };
        typedef NullType PointeeType;
    };
template <class U> struct PointerTraits<U*>
    {
        enum { result = true };
        typedef U PointeeType;
    };
public:
    typedef typename PointerTraits<T>::PointeeType PointeeType;
    typedef typename Select<isStdArith || isPointer || isMemberPointer,
        T, ReferredType&>::Result ParameterType;
    typedef typename UnConst<T>::Result NonConstType;
    ...
};
```

Pierwsza definicja wprowadza szablon klasy `PointerTraits`, który mówi: „`T` nie jest typem wskaźnikowym, a typ obiektu wskazywanego jest pusty” (`NullType` jest tutaj sygnalizatorem braku typu).

Druga definicja (z wierszem wyróżnionym pogrubieniem) wprowadza częściową specjalizację szablonu `PointerTraits`, pasującą do każdego typu wskaźnikowego. W przypadku jakichkolwiek wskaźników specjalizacja ta pasuje lepiej niż szablon ogólny, więc składowa `result` otrzymuje wartość `true`. Dodatkowo dla typów wskaźnikowych odpowiednio definiowany jest typ obiektu wskazywanego.

Możemy teraz zerknąć do wnętrza implementacji `std::vector::iterator`; wielu docieka, czy jest to zwyczajny wskaźnik, czy jakiś złożony obiekt?

```
int main()
{
    const bool
        iterIsPtr = TypeTraits<vector<int>::iterator>::isPointer;
    cout << "vector<int>::iterator jest " <<
        (iterIsPtr ? "szybki" : "inteligentny") << '\n';
}
```

Analogicznie `TypeTraits` implementuje stałe `IsReference` wraz z definicją `ReferencedType` dla typów referencyjnych. Dla typu referencyjnego `T` typ `ReferencedType` to typ, do którego odnosi się referencja do `T`; jeśli `T` jest typem prostym, `ReferencedType` jest po prostu typem `T`.

Wykrywanie wskaźników do składowych (omówienie wskaźników do składowych znajduje się w rozdziale 5.) wygląda nieco inaczej. Potrzebna jest inna specjalizacja, jak poniżej:

```
template <typename T>
class TypeTraits
{
private:
    template <class U> struct PToMTraits
    {
        enum { result = false };
    };
    template <class U, class V>
    struct PToMTraits<U V::*>
    {
        enum { result = true };
    };
public:
    enum { isMemberPointer = PToMTraits<T>::result };
    ...
};
```

2.10.2. Wykrywanie typów elementarnych

Szablon `TypeTraits<T>` implementuje stałą `IsFundamental`, ustawioną na `true` dla tych typów, które są typami elementarnymi. Do standardowych typów elementarnych zaliczymy typ `void` oraz wszystkie typy liczbowe (a więc całkowitoliczbowe i zmiennoprzecinkowe). Szablon `TypeTraits` definiuje też stałą określającą kategorię, do której należy dany typ.

Warto (kosztem wyprzedzenia omówienia) powiedzieć już teraz nieco o magii list typów (omawianych w rozdziale 3.) — znakomicie ułatwiają one wykrycie przynależności typu do pewnego określonego zestawu typów. Na razie wystarczy znajomość wyrażenia, które taką przynależność ustala:

```
TL::IndexOf<TYPELIST_nn(lista typów wymienionych po przecinku), T>::value
```

(gdzie *nn* jest liczbą typów na liście). Wyrażenie to zwraca indeksowaną od zera pozycję typu `T` na liście albo `-1`, jeśli typ `T` nie występuje na liście. Na przykład wyrażenie:

```
TL::IndexOf<TYPELIST_4(signed char, short int, int, long int), T>::value
```

będzie nieujemne tylko dla `T` będącego typem liczby całkowitej ze znakiem.

Oto definicja części szablonu `TypeTraits` odpowiadającej za wykrywanie typów elementarnych:

```
template <typename T>
class TypeTraits
{
    ... jak poprzednio ...
public:
    typedef TYPELIST_4(
```

```

    unsigned char, unsigned short int,
    unsigned int, unsigned long int)
    UnsignedInts;
typedef TYPELIST_4(signed char, short int, int, long int)
    SignedInts;
typedef TYPELIST_3(bool, char, wchar_t) OtherInts;
typedef TYPELIST_3(float, double, long double) Floats;
enum { isStdUnsignedInt =
    TL::IndexOf<UnsignedInts, T>::value >= 0 };
enum { isStdSignedInt = TL::IndexOf<SignedInts, T>::value >= 0 };
enum { isStdIntegral = isStdUnsignedInt || isStdSignedInt ||
    TL::IndexOf<OtherInts, T>::value >= 0 };
enum { isStdFloat = TL::IndexOf<Floats, T>::value >= 0 };
enum { isStdArith = isStdIntegral || isStdFloat };
enum { isStdFundamental = isStdArith || Conversion<T,
    void>::sameType };
    ...
};

```

Użycie list typów i wyrażenia `TL::IndexOf` daje możliwość szybkiego pozyskania informacji o typach bez konieczności wielokrotnego specjalizowania szablonu. Wszystkich, którzy nie mogą oprzeć się pokusie zajrzenia do wnętrza implementacji list typów i `TL::IndexOf`, zapraszam do lektury rozdziału 3. — byleby jednak tu wrócili.

Faktyczna implementacja wykrywania typów elementarnych jest nieco bardziej wyrafinowana, pozwalając również na wykrywanie typów rozszerzonych, definiowanych przez producentów poszczególnych implementacji — jak typy `int64` czy `long long`.

2.10.3. Optymalizacja typów parametrów funkcji

W kodzie szablonowym niekiedy potrzebna jest odpowiedź na pytanie: „Jaka jest najbardziej efektywna forma przekazywania i przyjmowania obiektów typu `T` w roli argumentów wywołania funkcji dla danego typu `T`?”. Zasadniczo w przypadku typów złożonych najbardziej efektywne jest przekazywanie przez referencję; typy proste (skalarnie) najlepiej przekazywać przez wartość (do typów skalarnych zaliczymy opisywane wcześniej elementarne typy liczbowe oraz wyliczenia, wskaźniki i wskaźniki do składowych). W przypadku typów złożonych przekazywanie przez referencję pozwala uniknąć narzutu związanego z wykonaniem kopii tymczasowej (z wywołaniami konstruktora i destruktoru), a dla typów skalarnych przy przekazywaniu przez wartość unika się narzutu odwołań pośrednich przez referencję.

Sęk w tym, że C++ nie pozwala na tworzenie referencji do referencji. Jeśli więc `T` jest już referencją, nie należy próbować dodawać do niego kolejnej referencji.

Odrobina analizy przy optymalizowaniu typów parametrów w wywołaniach funkcji prowadzi do ukucia poniższego algorytmu. Na jego potrzeby nazwiemy typ wartości przekazywanej do funkcji mianem `ParameterType`.

Jeśli `T` jest referencją do jakiegoś typu, `ParameterType` będzie identyczny z `T` (brak zmiany typu). *Przyczyna*: zakaz tworzenia referencji do referencji.

W przeciwnym razie:

Jeśli `T` jest typem skalarnym (`int`, `float` itd.), `ParameterType` będzie identyczny z `T` (brak zmiany typu). *Przyczyna*: typy elementarne najlepiej przekazywać przez wartość.

W przeciwnym razie `ParameterType` to `const T&`. *Przyczyna*: co do zasady, typy inne niż elementarne najlepiej przekazywać przez referencje.

Istotnym osiągnięciem tego algorytmu jest uniknięcie błędu utworzenia referencji do referencji, który mógłby wystąpić w przypadku prostego połączenia standardowych funkcji `bind2nd` i `mem_fun`.

Cechę `TypeTraits::ParameterType` można łatwo zaimplementować na bazie już gotowej techniki i na bazie już zdefiniowanych cech typów: `ReferencedType` i `IsFundamental`.

```
template <typename T>
class TypeTraits
{
    ... jak poprzednio ...
public:
    typedef Select<isStdArith || isPointer || isMemberPointer,
        T, ReferencedType&>::Result
        ParameterType;
};
```

Niestety, w tym układzie nie uda się zoptymalizować przekazywania przez wartość typów wyliczeniowych (enum).

Z cechy `TypeTraits::ParameterType` korzysta szablon klasy `Functor` z rozdziału 5.

2.10.4. Obieranie typu z kwalifikatorów

Dla danego typu `T` można łatwo uzyskać jego wariant dla wartości niemodyfikowalnych — wystarczy napisać `const T`. Ale operacja odwrotna, to znaczy odjęcie typowi kwalifikatora `const`, jest nieco trudniejsza. Analogicznie, niekiedy zachodzi potrzeba pozbycia się z typu kwalifikatora `volatile`.

Implementacja „usuwania `const`” jest stosunkowo prosta, znów na bazie częściowej specjalizacji szablonu:

```
template <typename T>
class TypeTraits
{
    ... jak poprzednio ...
private:
    template <class U> struct UnConst
    {
        typedef U Result;
    };
    template <class U> struct UnConst<const U>
    {
        typedef U Result;
    };
public:
    typedef UnConst<T>::Result NonConstType;
};
```

2.10.5. Zastosowania TypeTraits

Szablon `TypeTraits` daje dostęp do wielu interesujących danych. Przede wszystkim pozwala choćby poprzez proste złożenie technik prezentowanych w rozdziale zaimplementować procedurę kopiowania obiektów z użyciem optymalizacji dla wybranych typów (patrz podrozdział 2.10). `TypeTraits` można użyć do sprawdzenia cech typów iteratorów `InIt` i `OutIt`, interesujących pod kątem optymalizacji kopiowania; w połączeniu z szablonem `Int2Type` można efektywnie rozproprowadzić wywołanie do optymalizowanej funkcji `BitBlast` albo do klasycznej implementacji `Copy`.

```
enum CopyAlgoSelector { Conservative, Fast };
//procedura "klasyczna" działa dla dowolnych typów
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIt first, InIt last, OutIt result, Int2Type<Conservative>)
{
    for (; first != last; ++first, ++result)
        *result = *first;
    return result;
}
//procedura szybka działa tylko dla wskaźników do danych prostych
template <typename InIt, typename OutIt>
OutIt CopyImpl(InIt first, InIt last, OutIt result, Int2Type<Fast>)
{
    const size_t n = last-first;
    BitBlast(first, result, n * sizeof(*first));
    return result + n;
}
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result)
{
    typedef TypeTraits<InIt>::PointeeType SrcPointee;
    typedef TypeTraits<OutIt>::PointeeType DestPointee;
    enum { copyAlgo =
        TypeTraits<InIt>::isPointer &&
        TypeTraits<OutIt>::isPointer &&
        TypeTraits<SrcPointee>::isStdFundamental &&
        TypeTraits<DestPointee>::isStdFundamental &&
        TypeTraits<SrcPointee>::isStdFloat == TypeTraits<
            DestPointee>::isStdFloat &&
        sizeof(SrcPointee) == sizeof(DestPointee) ? Fast :
        Conservative };
    return CopyImpl(first, last, result, Int2Type<copyAlgo>());
}
```

Sama funkcja `Copy` nie jest specjalnie przepracowana, ale i tak dzieje się tu sporo ciekawych rzeczy. Wyliczenie `copyAlgo` wybiera pomiędzy implementacjami kopiowania. Logika wyboru prezentuje się następująco: jeśli oba iteratory są wskaźnikami, oba typy wskazywane są typami elementarnymi, typ wskazywany źródłowy i docelowy są oba liczbami całkowitymi albo oba liczbami zmiennoprzecinkowymi, wreszcie także typ wskazywany źródłowy jest tego samego typu co docelowy — wtedy można wybrać funkcję `BitBlast`. Jeśli więc w kodzie użytkowym napiszemy:

```
int* p1 = ...;
int* p2 = ...;
unsigned int* p3 = ...;
Copy(p1, p2, p3);
```

funkcja `Copy` wywoła (tak, jak powinna) szybką implementację kopiowania, mimo że typy źródłowy i docelowy są różne.

Wadą `Copy` jest to, że nie optymalizuje wszystkiego, co dałoby się zoptymalizować. Nie optymalizuje chociażby kopiowania prostych struktur C niezawierających wyłącznie składowych elementarnych — tak zwanych struktur POD (od ang. *plain old data* — *zwykle stare dane*). Tymczasem standard dopuszcza bajtowe kopiowanie struktur prostych; jedynie `Copy` nie potrafi wykryć „prostoty” struktury i zrealizuje dla nich wolniejszą procedurę kopiowania obiekt po obiekcie. Powinniśmy się tu uciec do klasycznych cech typów (oprócz `TypeTraits`). Na przykład:

```
template <typename T> struct SupportsBitwiseCopy
{
    typedef typename TypeTraits<T>::NonConstType NonConstType;
    enum { result = TypeTraits<T>::isFundamental };
};
template <typename InIt, typename OutIt>
OutIt Copy(InIt first, InIt last, OutIt result,
           Int2Type<true>)
{
    typedef typename TypeTraits<typename TypeTraits<
        InIt>::PointeeType>::UnqualifiedType SrcPointee;
    typedef typename TypeTraits<typename Typetraits<
        OutIt>::PointeeType>::UnqualifiedType DestPointee;
    enum { useBitBlast =
        TypeTraits<InIt>::isPointer &&
        TypeTraits<OutIt>::isPointer &&
        SupportsBitwiseCopy<SrcPointee>::result &&
        SupportsBitwiseCopy<DestPointee>::result &&
        Conversion<SrcPointee, DestPointee>::sameType || (
            TypeTraits<SrcPointee>::isStdFundamental &&
            TypeTraits<DestPointee>::isStdFundamental &&
            TypeTraits<SrcPointee>::isStdFloat ==
            TypeTraits<DestPointee>::isStdFloat &&
            sizeof(SrcPointee) == sizeof(DestPointee)) ? Fast : Conservative };
    return CopyImpl(.rst, last, result, Int2Type<useBitBlast>());
}
```

Teraz możemy już przygotować `Copy` na szybkie kopiowanie wybranych prostych struktur danych poprzez wyspecjalizowanie szablonu `SupportsBitwiseCopy` i umieszczenie w nim wartości `true`:

```
template<> struct SupportsBitwiseCopy<MyType>
{
    enum { result = true };
};
```


Kompletny zestaw cech typów implementowanych w szablonie `TypeTraits` z biblioteki `Loki` wymienia tabela 2.1.

TABELA 2.1.
Składowe `TypeTraits<T>`

Nazwa składowej	Rodzaj	Opis
<code>isPointer</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest wskaźnikiem.
<code>PointeeType</code>	typ	Typ wskazywany przez <code>T</code> , jeśli <code>T</code> jest wskaźnikiem, <code>NullType</code> w pozostałych przypadkach.
<code>isReference</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest referencją.
<code>ReferencedType</code>	typ	Typ, do którego odnosi się <code>T</code> , jeśli <code>T</code> jest referencją, lub <code>T</code> w pozostałych przypadkach.
<code>ParameterType</code>	typ	Najbardziej odpowiedni typ do przekazywania obiektów typu <code>T</code> do wywołań funkcji niemodyfikujących (albo <code>T</code> , albo <code>const T&</code>).
<code>isConst</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest typem z kwalifikacją <code>const</code> (typem wartości niemodyfikowalnej).
<code>NonConstType</code>	typ	Typ <code>T</code> pozbawiony kwalifikatora <code>const</code> (jeśli taki występował).
<code>isVolatile</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest typem z kwalifikacją <code>volatile</code> (typem wartości ulotnej).
<code>NonVolatileType</code>	typ	Typ <code>T</code> pozbawiony kwalifikatora <code>volatile</code> (jeśli taki występował).
<code>NonQualifiedType</code>	typ	Typ <code>T</code> pozbawiony kwalifikatorów <code>const</code> i <code>volatile</code> (jeśli takie występowały).
<code>isStdUnsignedInt</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest jednym z czterech typów całkowitych bez znaku (<code>unsigned char</code> , <code>unsigned short</code> , <code>unsigned int</code> , <code>unsigned long</code>).
<code>isStdSignedInt</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest jednym z czterech typów całkowitych ze znakiem (<code>signed char</code> , <code>signed short</code> , <code>signed int</code> , <code>signed long</code>).
<code>isStdIntegral</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest standardowym typem liczbowym.
<code>isStdFloat</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest standardowym typem zmiennoprzecinkowym (<code>float</code> , <code>double</code> lub <code>long double</code>).
<code>isStdArith</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest standardowym typem arytmetycznym (całkowitym lub zmiennoprzecinkowym).
<code>isStdFundamental</code>	stała logiczna	Wartość <code>true</code> , jeśli <code>T</code> jest typem elementarnym (jednym z typów arytmetycznych lub <code>void</code>).

2.11. Podsumowanie

Komponenty projektowe omawiane w tej książce są budowane z cegiełek, z których część stanowi samodzielne techniki programistyczne. Większość z nich przydaje się programistom szablonów. W tej roli omawialiśmy:

- *Asercje czasu kompilacji* (podrozdział 2.1), pomocne w bibliotekach chcących prezentować znaczące komunikaty o błędach z kodu szablonowego.
- *Częściowe specjalizacje szablonów* (podrozdział 2.2), pozwalające na specjalizowanie szablonu nie tylko dla kompletnego zestawu wartości parametrów, ale także dla ich podzbiorów, a więc dla całych rodzin wartości parametrów pasujących do wzorca specjalizacji.
- *Klasy lokalne* (podrozdział 2.3), ciekawe zwłaszcza wewnątrz szablonów funkcji.

- *Odwzorowania stałych całkowitych na typy* (podrozdział 2.4), ułatwiające statyczne rozprowadzanie wywołań na bazie wartości liczbowych (najczęściej — wartości warunków logicznych).
- *Odwzorowania typów na typy* (podrozdział 2.5), pozwalające na zastąpienie niedozwolonej w C++ częściowej specjalizacji szablonów funkcji przeciążaniem funkcji.
- *Selekcję typów* (podrozdział 2.6), pozwalającą na statyczne wybieranie typów na bazie wartości logicznych.
- *Statyczne wykrywanie relacji dziedziczenia i możliwości konwersji* (podrozdział 2.7), dające możliwość sprawdzenia, czy dwa dowolne typy pozostają ze sobą w relacji dziedziczenia, czy można wykonać konwersję pomiędzy nimi i czy czasem nie są one tożsame.
- *Szablon `TypeInfo`* (podrozdział 2.8) jako otoczka klasy `std::type_info`, dająca obiektom `type_info` semantykę wartości i możliwość łatwego porównywania.
- *Klasy `NullType` i `EmptyType`* (podrozdział 2.9) jako przydatne typy zastępcze w metaprogramowaniu szablonowym.
- *Szablon `TypeTraits`* (podrozdział 2.10), udostępniający cały szereg uniwersalnych cech dowolnych typów do łatwego wykorzystania przy dostosowywaniu kodu szablonowego do możliwości poszczególnych kategorii typów.

Skorowidz

#ifdef, dyrektywa preprocesora, 165
_DestroySingleton, 161
_buffer, 161

A

Abstract Factory, 71–73, 247–263
AbstractEnemyFactory, 249–255
AbstractFactory, 247–263
AbstractFactoryUnit, 251–253
AbstractProduct, 236–245
Accept, 270, 283
AcceptImpl, 281, 285, 288
ACE (Adaptive Communication Environment), 342
Acquire, 190, 338,339
Action, 124
active commands, 126
Adapter, 315
Add, 308–313, 319, 323-325
after, 37
algorytm
 copy, 61
 czasu kompilacji, 98
 operacji na listach typów, 98
 przeszukiwanie liniowe, 79
Allocate, 106, 107
allocChunk_, 109, 111
AllowConversion, 222
alokacje na stercie, 149
ALU (jednostka arytmetyczno logiczna), 336
AnotherDerived, 225
API (interfejs programistyczny), 189
Aplikacja, 124
Append, 80, 98
argumenty, konwersja typów, 139, 316–320
ArrayStorage, 220, 224
asercja, czas kompilacji, 43–46
assert, 211

AssertCheck, 223
AssertCheckStrict, 223
AssocVector, 239
atexit, 161, 165, 169
ATEXIT_FIXED, 166
AtExitFn, 172
AtomicAdd, 337
AtomicAssign, 337
AtomicDecrement, 215
AtomicIncrement, 215
auto_ptr, 132, 133, 153
available_, 103

B

backEnd_, 312
BackEndType, 314
BadMonster, 249
BadSoldier, 248
BadSuperMonster, 248
BankAccount, 338, 339
Bar, 166
Base, 85
BaseLhs, 299–331
BaseProductList, 255
BaseRhs, 299, 329, 331
BaseSmartPtr, 26
BaseVisitable, 283, 290, 291
BaseVisitor, 274, 279–283
BaseVisitorImpl, 290, 291
BasicDispatcher, wytyczna, 307–314, 322–330
BasicFastDispatcher, wytyczna, 322–330
before, 37
BinderFirst, 146
BindFirst, 145, 153
binding, 144
BitBlast, 61, 66

blocksAvailable_, 106, 107
 blockSize_, 110
 błędy
 asercja, 44
 interfejs wszechstronny, 24
 raportowanie, 210
 Button, 72, 247
 byty funkcyjne, 128

C

callable entities, 128
 callback, 127
 callbackMap_, 310, 312
 callbacks_, 233, 323
 CallbackType, 307, 310, 330
 CastingPolicy, wytyczna, 318–330
 CatchAll, wytyczna, 288–291
 Chain, 147
 char, 45, 58, 139
 CheckingImpl, 223
 CheckingPolicy, 36, 224
 chunk, 105–108
 chunkSize, 112
 Circle, 231
 ClassLevelLockable, 181, 339, 342
 Clone, 50, 149, 192, 221, 222, 240
 clone, wytwórnia obiektów, 240
 CloneFactory, 242, 246
 CLOS, 293
 COM, 160, 235
 Command, 124–126
 CompileTimeChecker, 45
 CompileTimeError, 45
 COMRefCounted, 222, 224
 ConcreteCommand, 124, 126
 ConcreteFactory, 254, 261–263
 ConcreteLifetimeTracker, 172
 ConcreteProduct, 255
 const, 92, 128, 140, 148, 193, 194, 235, 337
 ConventionalDialog, 247
 Conversion, wytyczna, 56–58, 64, 67, 218, 222
 Copy, 61, 66
 copy_backward, 171
 copyAlgo, 66
 CORBA (Common Object Request Broker
 Architecture), 140, 160
 Create, 28, 29, 32, 35, 52, 53, 73, 226
 CreateButton, 72
 CreateDocument, 227
 CreateObject, 237
 CreateScrollbar, 72
 CreateShape, 234
 CreateShapeCallback, 232
 CreateStatic, 181

CreateT, 251
 CreateUsingMalloc, 181
 CreateUsingNew, 181
 CreateWindow, 72
 CreationPolicy, 179
 Creator, 28–34, 177–181
 czas kompilacji
 asercja, 43–46
 wykrywanie dziedziczenia, 55

D

Deallocate, 106–112
 deallocChunk_, 111
 DeepCopy, 222
 DEFAULT_CHUNK_SIZE, 117, 120
 DEFAULT_THREADING, 118
 DefaultLifetime, 181
 DEFINE_CYCLIC_VISITABLE, 286
 DEFINE_VISITABLE, 283, 291
 delete, 187, 212
 DeleteChar, 147
 Deposit, 338
 Derived, 115, 225, 228
 DerivedToFront, 85, 98, 303
 Destroy, wytyczna, 41
 destroyed_, 162–164
 DestructiveCopy, 221, 224
 destruktor klas wytycznych, 33
 detekcja, martwe referencje, 162
 Dialog, 247
 DisallowConversion, 222
 DispatcherBackend, wytyczna, 325–330
 DispatchRhs, 300
 Display, 164, 167, 173, 198
 DisplayStatistics, 270
 DocElement, 265–277, 288
 DocElementVisitor, 265–277
 DocElementVisitor.h, 273
 DoClone, 241
 DoCreate, 252, 254, 260
 DocStats, 269, 270, 277
 Document, 226
 DocumentManager, 227
 dostęp swobodny, 77
 DottedLine, 241
 double dispatch, 293, 307, 328
 Double-Checked Locking, wzorzec projektowy,
 174–176
 DoubleDispatch, 297, 298
 Drawing, 229, 231
 DrawingDevices, 321
 DrawingType, 231
 Dylan, 293, 294
 dynamic_cast, 268, 274–279, 284, 288

DynamicCaster, 319, 330
dziedziczenie
 wykrywanie statyczne, 55

E

EasyLevelEnemyFactory, 257, 259, 263
ElementAt, 41
Ellipse, 231, 298
else, 268
EmptyType, 60, 69
EnforceNotNull, 36, 39
enum, 65
erase, 308
Erase, 81, 83, 98
EraseAll, 81, 83, 98
EventHandler, 94
Execute, 124, 126
Executor, 299, 329
ExtendedWidget, 39, 192

F

Factory, 236, 237, 243
FactoryError, wytyczna, 237, 238
FactoryErrorImpl, 237
FactoryErrorPolicy, 245, 246
FastWidgetPtr, 38
Field, 90, 92, 99
Fire, 300, 302, 305, 329
FixedAllocator, 104, 108–112, 119
FnDispatcher, 311, 312, 316, 324
forwarding command, 126
free, 170, 220
fun_, 138, 140
FunctorDispatcher, 314, 315, 319, 324, 330
FunctorHandler, 135–139
FunctorImpl, 129–136, 149, 152
FunctorType, 314

G

GameApp, 258
GenLinearHierarchy, 254–258, 261
GenScatterHierarchy, 87–92, 251, 254, 261
geronimosWork, 142
GetClassIndex, 323
GetClassIndexStatic, 323
GetImpl, 191, 202, 213, 224
GetImplRef, 191, 210
GetLongevity, 181, 182
GetPrototype, 29, 32, 35
głęboka kopia, 149, 199, 222
GraphicButton, 84
GUI (graphical user interface), 126

H

handle, 189
HatchingDispatcher, 304
HatchingExecutor, 302
HatchRectanglePoly, 309
HeapStorage, 220, 224
hierarchie
 liniowe, 84
 rozrzucone, 87
HTMLDocument, 227

I

IdToProductMap, 242
if, instrukcja, 268
if-else, instrukcja, 50, 297, 299
IMPLEMENT_INDEXABLE_CLASS, 322
IncrementFontSize, 270
IndexOf, 98
INHERITS, 58
Inicjalizacja leniwa, 211
InIt, 61
insert, 233
InsertChar, 145, 151
Instance, 167, 173, 174, 179
Int2Type, 43, 48, 49, 51, 66, 91, 92
inteligentny wskaźnik, 24, 25, 36, 185
 głębokie kopie, 191
 goły (raw), 200
 kontrola przy inicjacji, 210
 kontrola przy wyłuskaniu, 211
 kopiowanie przy zapisie, 193
 kopiowanie z usuwaniem, 197
 metody, 190
 operator pobrania adresu, 199
 raportowanie błędów, 210
 równość, nierówność, 202
 wiązanie odwołań, 196
 wielowątkowość, 213
 zarządzanie posiadaniem, 191
interfejs wszechstronny, błędy, 24
IntType, 215, 336
InvocationTraits, 305
isConst, 68
isPointer, 68
isReference, 62, 68
isStdArith, 68
isStdFloat, 68
isStdFundamental, 68
isStdIntegral, 68
isStdSignedInt, 68
isStdUnsignedInt, 68
isVolatile, 68

K

KDL, problem, 162–64
 keyboard, 162–64
 KillPhoenixSingleton, 165
 klasa
 blokowanie, 216
 dekomponowanie, 37
 finalna, 49
 generowanie z list typów, 87–97
 kliencka, 181
 lokalna, 48
 pochodna, 255
 klasy, wytwórnice obiektów, 228
 klawiatura, 162–164
 kolekcja asocjacyjna, 232
 komendy
 żądania aktywne, 126
 żądania delegowane, 126
 konwersja
 argumentów, 139, 316–20
 niejawna, typ gołego wskaźnika, 200
 wartości zwracanej, 139
 wiązanie argumentów, 144
 własna, 200
 kopia głęboka, 217
 kowariancja typów zwracanych, 240

L

lazy initialization, 211
 Length, 76, 77, 98
 less, 209
 Lifetime, wytyczna, 169–183
 LifetimeTracker, 172
 LifeTimeTracker, 169
 LIFO (last in, first out), 161
 LISP, 75
 ListOfTypes, 326
 listy typów
 definiowanie, 73
 dopisywanie, 80
 dostęp swobodny, indeksy, 77
 generowanie klas, 87
 indeksy, 77
 obliczanie długości, 76
 podstawy, 71
 porządkowanie, 84
 przeszukiwanie, 79
 zastępowanie elementów, 83
 tworzenie liniowe, 75
 usuwanie, 81
 usuwanie duplikatów, 82
 zastosowanie, 71

Lock, 174, 213, 339
 LockedStorage, 220
 LockingProxy, 214
 Log, 162–168
 logic_error, 180
 Loki, 73–119, 239–342
 longevity control, 167
 lower_bound, 307

M

MacroCommand, 126, 147
 MakeAdapter, 48
 MakeCopy, 193
 MakeT, 251
 malloc, 170, 181
 małe obiekty, przydział pamięci
 alokator przydziałów o stałym rozmiarze, 108
 alokator, zasada działania, 102
 chunk, 105
 domyślny alokator, 102
 podstawy, 101
 mapa, 209
 mapowanie
 typ na typ, 48, 53
 wartość na typ, 43, 48–51, 66
 mapy, 232, 327
 martwe referencje, problem, 162–164
 MAX_SMALL_OBJECT_SIZE, 117, 118
 maxObjectSize, 112
 MemControlBlock, 102
 MemFunHandler, 142
 menedżer zależności, 168
 ML, 293
 ModalDialog, 47
 Monster, 247, 253
 MostDerived, 85, 98
 MultiThreaded, 25, 26
 muteks, 174, 177, 337, 339
 MyController, 47
 MyOnlyPrinter, 157
 MyVisitor, 285, 286

N

narzut czasu wykonania, 266
 next_, 216
 NiftyContainer, 50, 53
 NoCheck, 223
 NoChecking, 36, 39
 NoCopy, 222
 NoDestroy, 181
 NoDuplicates, 82, 83, 98
 NonConstType, 68
 NonVolatileType, 68
 NullType, 60, 62, 69, 74, 76, 80, 98, 300

O

ObjectLevelLockable, 339–341
 odwołania, zliczanie, 194
 Okno, 47, 48, 72, 73, 93–96, 125, 147
 OnDeadReference, 163, 164, 180
 OnError, 302, 329
 OnEvent, 93
 OnUnknownVisitor, 289, 291
 operator
 pobrania adresu, 199
 porównania, 202–207
 przydziału, 165
 OpNewCreator, 30
 OpNewFactoryUnit, 254, 255
 OrderedTypeInfo, 246
 ortogonalne, wytyczne, 41
 Outfit, 61
 Ownership, wytyczna, 199–222

P

pamięć
 alokator, zasada działania, 102–104
 chunks, 105–108
 RMW (read-modify-write), wczytanie-
 modyfikacja-zapis, 336
 Paragraph, 267, 270, 272, 275, 276, 283
 ParagraphVisitor, 275, 276, 280
 ParameterType, 64, 68
 ParentFunctor, 136
 Parrot, 142, 143
 pData_, 110
 pDocElem, 276
 pDuplicateShape, 241
 pDynObject, 167
 pFactory_, 250
 Phoenix Singleton, wzorzec projektowy, 164
 pimpl, 102
 pInstance_, 158, 163, 173, 175, 179, 181
 placement new, 165
 pLastAlloc_, 113
 plik nagłówkowy
 DocElementVisitor.h, 273
 SmallAlloc.h, 120
 Typelist.h, 73, 75
 POD (plain old data), struktury, 67
 podwójne przełączanie, 296–298
 podziału czasu, 333
 Point3D, 93
 pointee_, 186, 188, 189, 197, 207, 212, 219
 PointeeType, 68
 PointerToObj, 143
 PointerTraits, 62
 PointerType, 189, 220

polimorfizm, 101, 118, 192
 Polimorfizm, 102
 Poly, 298, 305, 310
 Polygon, 231, 240
 prev_, 216
 Printer, 190
 printf, 130
 printingPort_, 157
 priority_queue, 169
 ProductCreator, 239, 242, 245, 246
 produkt abstrakcyjny, 235–245, 250, 259
 Prototype, 32, 256–261
 PrototypeFactoryUnit, 259, 260, 262
 przydział pamięci
 domyślny alokator, 102
 małe obiekty, 101–120
 chunk, 105–108
 o stałym rozmiarze, 108–11
 swobodne przydzielanie i zwalnianie, 110
 przydziały masowe, 110
 zasada działania, 102–104
 zwalnianie sekwencyjne, 110
 pTrackerArray, 170

R

race condition, sytuacja hazardowa, 173
 RasterBitmap, 278, 279, 285
 RasterBitmapVisitor, 280
 realloc, 170
 Receiver, 124, 126
 Rectangle, 297
 redo, 145
 RefCounted, 26, 222, 224
 RefCountedMT, 222
 reference linking, 196
 ReferencedType, 62, 65, 68
 RefLinked, 222, 224
 RegisterShape, 232–234
 RejectNull, 223, 224
 RejectNullStatic, 223, 224
 RejectNullStrict, 223, 224
 Release, 105, 106, 190, 191, 221, 338, 339
 Replace, 83, 84, 86, 98
 ReplaceAll, 84, 98
 Reset, 191
 Result, 54, 62, 65, 78, 80, 81–86, 90, 92, 98, 133,
 256, 285, 303
 ResultType, 129–131, 135, 136, 142, 143, 145, 146,
 298–301, 306–315, 319, 322, 324–331
 RMW (read-modify-write), wczytanie-
 modyfikacja-zapis, 336
 RoundedRectangle, 297, 298, 302, 309, 316, 317, 318
 RoundedShape, 316, 317

rozprowadzanie podwójne, 293, 307, 328
 równość, 202–207
 różność, 202–207

S

safe_reinterpret_cast, 44, 46
 SafeWidgetPtr, 35, 38
 sameType, 57, 58, 64, 67
 Save, 230
 ScheduleDestruction, 177–180
 Scroll, 147
 ScrollBar, 72, 82, 84, 94, 96
 Secretary, 26
 Select, 43, 54, 62, 65, 85, 86
 SetLongevity, 167–172, 180
 SetPrototype, 29, 32, 34, 260
 Shape, 229–235, 240–244, 295–298, 301–305,
 309–320, 326, 329
 ShapeFactory, 232–234, 243, 244
 signed char, 68
 SillyMonster, 248–250, 254, 258, 259, 263
 SillySoldier, 248–250, 254, 258, 259, 263
 SillySuperMonster, 248, 249, 254, 258, 259, 263
 SingleThreaded, 35, 178–183, 340–342
 singleton
 dekompozycja, 176
 martwe referencje, problem, 162
 Meyersa, 160
 podstawowe idiomy c++, 157
 podstawy, 155
 usuwanie, 159
 wielowątkowość, 173
 żywołność, 169
 SingletonWithLongevity, 182
 size_t, 57, 61, 66, 102–108, 112, 114, 116, 118–120
 SmallAlloc.h, 120
 SmallObjAllocator, 105, 112–119
 SmartPtr, 23, 27, 35–40, 111, 185–224
 Soldier, 247, 249, 250, 252, 253, 255, 257, 258, 262
 SomeLhs, 300, 306, 308, 309, 314, 315, 319, 323,
 325, 330, 331
 SomeRhs, 306, 308, 309, 314, 315, 319, 323, 325,
 330, 331
 SomeThreadingModel, 335, 337
 SomeVisitor, 279, 283
 splmpl_, 132, 134, 136, 143, 148, 149
 static_cast, 90, 108, 138, 171, 172, 316, 318–320, 330
 STATIC_CHECK, 44–46
 StaticDispatcher, 298–306, 312, 328–330
 sterowania żywotnością, 167
 STL, 59, 110, 140, 200, 239
 Storage, wytyczna, 37, 38, 189–223
 StorageImpl, 219

Strategy, wzorzec projektowy, 28
 String, 53, 334, 339
 SuperMonster, 247–262
 SUPERSUBCLASS, 57, 58, 85, 86
 Surprises.cpp, 253
 SwitchPrototype, 34, 35
 sytuacja hazardowa, 173
 szablon Functor, 126, 138, 150

T

tablice, 41, 212
 obiektów, 41
 Tester, 206
 Tester*, 206
 ThreadingModel, wytyczna, 35–38, 116–183, 215,
 335, 336–342
 time slicing, 333
 typ kliencki, 181
 type_info, 43, 58–60, 69, 76, 235, 242–246, 307,
 326, 327
 Type2Type, 43, 48, 52, 53, 91, 92, 251, 252, 255,
 260, 262
 TypeAt, 77, 78, 81, 98
 TypeAtNonStrict, 78, 98, 133
 TypeInfo, 43, 58–60, 69, 242, 243, 246, 307, 308, 326
 Typelist, 63–98, 131–153, 252–330
 Typelist.h, 73, 75
 TypesLhs, 298–306, 328, 329
 TypesRhs, 298–306, 328, 329
 TypeTraits, 62–69, 148, 149, 212

U

UCHAR_MAX, 107
 uchwyt, 189
 Undo, 151
 Unlock, 213, 214
 unsigned char, 68
 UpdateStats, 267, 268
 upper_bound, 171

V

ValueType, 54, 55
 VectorGraphic, 274
 VectorizedDrawing, 288
 Visitable, 278
 Visitor, wzorzec projektowy, 265–289
 podstawy, 265
 przeciążenia, 271
 wariacje, 287
 wizytacja acykliczna, 273
 wizytacja wybiórcza, 289

VisitParagraph, 269–277
 VisitRasterBitmap, 269–271, 276
 VisitVectorGraphic, 274
 VolatileType, 178, 179, 341

W

wektor, 196
 wiązanie

- argumentów, 144
- odwołań, 196

 Widget, 27–58, 82–90, 186–214, 337, 341
 WidgetEventHandler, 94–96
 WidgetFactory, 72, 73
 WidgetInfo, 88–92
 WidgetManager, 29–40
 wielometody, 293

- metoda siłowa, 296
- podwójne przełączanie, 307–312
- przydatność, 295
- symetria, 303

 wielowątkowość, 173–175, 333

- biblioteka, 333
- inteligentne wskaźniki, 213
- muteks, 216
- zliczanie odwołań, 215

 Window, 47, 48, 72, 73, 93, 96, 125, 147
 wirtualny konstruktor, dylemat, 256
 Withdraw, 338
 wyjątki, 238
 wykonanie asynchroniczne, 334
 wyluskanie, kontrola, 211
 wytwórnice obiektów

- implementacja, 229
- klasy, 228
- klony, 240
- podstawy, 225
- produkt abstrakcyjny, 235

 wytyczna

- BasicDispatcher, 307–314, 322–330
- BasicFastDispatcher, 322–330
- CastingPolicy, 318–330

CatchAll, 288–291
 CheckingPolicy, 34, 224
 Conversion, 56–67, 218, 222
 CreationPolicy, 179
 Creator, 28–34, 177–181
 Destroy, 41
 DispatcherBackend, 325–330
 FactoryError, 237, 238
 Lifetime, 169–183
 Ownership, 199–222
 Storage, 37, 38, 189–223
 tablica obiektów, 41
 ThreadingModel, 35–38, 116–183, 215, 335, 336–342
 wytyczne klas

- destruktor, 33
- implementacja, 30
- konfigurowanie, 37
- łączenie wytycznych, 35
- podstawy, 23–42

 wywołanie zwrotne, 127, 233, 311
 wzorzec projektowy

- Abstract Factory, 71–73, 247–263
- Command, 124–26
- Double-Checked Locking, 174–176
- Phoenix Singleton, 164
- Prototype, 32, 256–261
- Strategy, 28
- Visitor, 265–289

X

X Windows, 128

Z

zarządzanie posiadaniem, strategię, 191–199
 zasoby, efektywne wykorzystanie, 334
 zdarzenia, 93, 342

Ż

żądania skomasowane, 147

NOWOCZESNE PROJEKTOWANIE W C++

Uogólnione implementacje
wzorców projektowych

KANON INFORMATYKI

Język C++ jest obecny na rynku już niemal trzydzieści lat, a jednak nadal świetnie spełnia swoje zadania. Jest powszechnie używany, a wręcz niezastąpiony w wielu dziedzinach programowania. Wszędzie tam, gdzie potrzebna jest najwyższa wydajność oraz pełna kontrola nad zasobami i przebiegiem programu, sprawdza się wysmienicie. Wystarczy odrobina chęci, dobry podręcznik i trochę czasu, aby wykorzystać pełną moc C++ w nowoczesnych technikach programowania.

Książkę, która Ci w tym pomoże, trzymasz właśnie w rękach. Czy znajdziesz czas i ochotę, aby zgłębić zawartą w niej wiedzę? Gwarantujemy, że warto! W trakcie lektury dowiesz się, jak zaimplementować w C++ najpopularniejsze wzorce projektowe. Dzięki nim błyskawicznie oprogramujesz typowe rozwiązania. Nauczysz się tworzyć dokładnie jedną instancję obiektu oraz zobaczysz, jak korzystać z fabryki obiektów czy inteligentnych wskaźników. Ponadto zapoznasz się z technikami projektowania klas, asercjami w trakcie kompilacji oraz uogólnionymi funkcjami. Dzięki tej książce poczujesz na nowo satysfakcję z pisania programów w języku C++!

Czerp satysfakcję z korzystania z nowoczesnych technik programowania w C++!

- Projektowanie klas
- Asercje czasu kompilacji
- Listy typów
- Alokowanie małych obiektów
- Funktory uogólnione
- Inteligentne wskaźniki
- Fabryka obiektów i fabryka abstrakcyjna
- Tworzenie dokładnie jednego obiektu — wzorzec singleton
- Multimetody

Nr katalogowy: 6301



Księgarnia internetowa:
<http://helion.pl>



Zamówienia telefoniczne:
0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje
① <http://helion.pl/promocje>
Książki najchętniej czytane
② <http://helion.pl/bestsellery>
Zamów informacje o nowościach
③ <http://helion.pl/nowosci>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 88 63
e-mail: helion@helion.pl
<http://helion.pl>

helion.pl
księgarnia
internetowa

Cena 69,00 zł

ISBN 978-83-246-3301-2



9 788324 633012

Informatyka w najlepszym wydaniu