Wprowadzenie

Jeśli jesteś doświadczonym programistą C++ i choć trochę mnie przypominasz, początkowo pomyślisz o języku C++11 „Tak, tak, wiem. To C++, tylko trochę bardziej". Jednak, gdy dowiesz się więcej, będziesz zaskoczony zasięgiem zmian. Deklaracje `auto`, pętle `for` oparte na zakresie, wyrażenia lambda i odwołania do r-wartości zmieniają oblicze języka C++, nie mówiąc nic o nowych funkcjach współbieżności. Następnie pojawiają się zmiany idiomatyczne. Znikają wartości `0` i definicje typów `typedef`, pojawiają się wskaźniki `nullptr` i deklaracje aliasów. Wyliczenia powinny teraz mieć określony zasięg. Wskaźniki inteligentne są teraz preferowane względem wbudowanych. Przenoszenie obiektów jest zwykle lepsze niż ich kopiowanie.

Jest dużo do nauczenia się o C++11, nie wspominając o C++14.

Ważniejsze jest to, że jest dużo do nauczenia się o *skutecznym* korzystaniu z nowych możliwości. Jeśli potrzebujesz podstawowych informacji o „nowoczesnych" funkcjonalnościach języka C++, zasoby są obfite, ale jeżeli szukasz wskazówek, jak użyć tych funkcjonalności do tworzenia oprogramowania, które będzie poprawne, wydajne, łatwe w utrzymaniu i przenośne, znalezienie ich będzie znacznie trudniejsze. Tu przyda się niniejsza książka. Nie jest poświęcona opisaniu funkcjonalności C++11 i C++14, ale ich skutecznemu stosowaniu.

Informacje w tej książce są podzielone na wskazówki nazywane *punktami*. Chcesz zrozumieć różne formy dedukcji typów? Albo dowiedzieć się, kiedy (a kiedy nie) używać deklaracji `auto`? Interesuje Cię, dlaczego funkcje składowe `const` powinny być bezpieczne wątkowo, jak zaimplementować idiom Pimpl przy użyciu wskaźnika `std::unique_ptr`, dlaczego należy unikać domyślnych trybów przechwytywania w wyrażeniach lambda lub czym różnią się typy `std::atomic` i `volatile`? Znajdziesz tu wszystkie odpowiedzi. Co więcej, są one niezależne od platformy i zgodne ze standardami. Jest to książka o *przenośnym* języku C++.

Punkty w tej książce są wskazówkami, a nie regułami, ponieważ wskazówki mają wyjątki. Najważniejszą częścią każdego punktu nie jest oferowana rada, ale jej uzasadnienie. Po przeczytaniu, będziesz móc wyznaczyć, czy okoliczności danego

projektu uzasadniają naruszenie wskazówki z danego punktu. Prawdziwym celem tej książki nie jest nakazanie, co robić, a czego unikać, ale przekazanie głębokiej wiedzy na temat sposobu działania języków C++11 i C++14.

Terminologia i konwencje

W celu zapewnienia, że się rozumiemy, ważne jest, abyśmy ustalili pewną terminologię, zaczynając ironicznie od samego „C++”. Istnieją cztery oficjalne wersje języka C++, każda nazwana od roku przyjęcia odpowiedniego standardu ISO: *C++98*, *C++03*, *C++11* i *C++14*. C++98 i C++03 różnią się jedynie szczegółami technicznymi, więc w tej książce będę się odnosił do nich jako do C++98. Gdy odnoszę się do C++11, mam na myśli zarówno C++11, jak i C++14, ponieważ C++14 jest faktycznie nadzbiorem C++11. Gdy piszę C++14, mam na myśli dokładnie C++14. A gdy po prostu piszę o C++, jest to zdanie ogólne dotyczące wszystkich wersji języka.

Używany termin	Wersja języka
C++	Wszystkie
C++98	C++98 i C++03
C++11	C++11 i C++14
C++14	C++14

W efekcie mogę powiedzieć, że język C++ stawia na pierwszym miejscu wydajność (prawdziwe dla wszystkich wersji), że w języku C++98 brakuje obsługi współbieżności (prawdziwe tylko dla C++98 i C++03), że język C++11 obsługuje wyrażenia lambda (prawdziwe dla C++11 i C++14) oraz że język C++14 oferuje uogólnioną dedukcję typu zwracanego przez funkcję (prawdziwe tylko dla C++14).

Najbardziej rozpowszechnioną funkcjonalnością C++11 jest prawdopodobnie semantyka przenoszenia, a podstawą semantyki przenoszenia jest rozróżnianie wyrażeń, które są *r-wartościami*, od tych, które są *l-wartościami*. Jest tak, ponieważ r-wartości wskazują obiekty podatne na operacje przenoszenia, a l-wartości – zasadniczo nie. Koncepcyjne (chociaż w praktyce nie zawsze) r-wartości odpowiadają tymczasowym obiektom zwracanym przez funkcje, a l-wartości odpowiadają obiektom, do których możemy się odwoływać przez nazwę, wskaźnik lub odwołanie do l-wartości.

Przydatną heurystyką pozwalającą wyznaczyć, czy wyrażenie jest l-wartością, jest zapytanie, czy możemy wziąć jej adres. Jeśli możemy, zwykle jest. Jeśli nie możemy, zwykle jest r-wartością. Przyjemną cechą tej heurystyki jest to, że pozwala zapamiętać, że typ wyrażenia jest niezależny od tego, czy wyrażenie jest l-wartością, czy r-wartością. Oznacza to, że dla danego typu `T` możemy mieć l-wartości typu `T`, a także r-wartości typu `T`. Jest to szczególnie ważne, aby pamiętać o tym, gdy mamy do czynienia z parametrem, którego typem jest odwołanie do r-wartości, ponieważ sam parametr jest l-wartością:

```
class Widget {
public:
  Widget(Widget&& rhs);    // rhs jest l-wartością, chociaż ma
  …                        // typ odwołania do r-wartości
};
```

Tutaj byłoby doskonale poprawne wzięcie adresu parametru `rhs` wewnątrz konstruktora przenoszącego obiektu `Widget`, więc `rhs` jest l-wartością, nawet jeśli jego typem jest odwołanie do r-wartości. (Dzięki podobnemu rozumowaniu wszystkie parametry są l-wartościami).

Ten fragment kodu demonstruje kilka używanych przeze mnie konwencji:

- Nazwą klasy jest `Widget`. Używam nazwy `Widget` zawsze, kiedy chcę określić arbitralny typ definiowany przez użytkownika. O ile nie potrzebuję pokazać konkretnych szczegółów klasy, używam nazwy `Widget` bez deklaracji.
- Używam nazwy parametru `rhs` („right-hand side” – prawostronny). Jest to preferowana przeze mnie nazwa parametru dla *operacji przenoszenia* (czyli konstruktora przenoszącego i przenoszącego operatora przypisania) i *operacji kopiowania* (czyli konstruktora kopiującego o kopiującego operatora przypisania). Korzystam z niej także dla parametrów prawostronnych operatorów binarnych:

  ```
  Matrix operator+(const Matrix& lhs, const Matrix& rhs);
  ```

  Mam nadzieję, że nie jest zaskakujące, że `lhs` oznacza lewostronny („left-hand side”).

- Stosuję specjalne formatowanie do części kodu lub części komentarzy, aby przyciągnąć do nich uwagę. W powyższym konstruktorze przenoszącym klasy `Widget` wyróżniłem deklarację `rhs` i część komentarza wskazującą, że `rhs`

to l-wartość. Wyróżniony kod nie jest z założenia dobry ani zły. Jest to po prostu kod, na który chcę zwrócić szczególną uwagę.

- Używam znaku „…”, aby pokazać, że „tu może znajdować się inny kod”. Ten wąski wielokropek jest odmienny od szerokiego wielokropka („ . . . ”), używanego w języku C++11 w kodzie źródłowym szablonów o zmiennej liczbie parametrów. Brzmi to mętnie, ale tak nie jest. Na przykład:

```
template<typename... Ts>                // to są wielokropki
void processVals(const Ts&... params)   // w kodzie źródłowym
{                                       // języka C++

  …                                     // to oznacza
                                        // „tu znajduje się inny kod”
}
```

  Deklaracja funkcji `processVals` pokazuje, że używam `typename`, gdy deklaruję parametry typu w szablonach, ale jest to zaledwie osobista preferencja. Słowo kluczowe `class` działałoby równie dobrze. W tych sytuacjach, gdy pokazuję urywki kodu ze standardu C++, deklaruję parametry typu przy użyciu słowa kluczowego `class`, ponieważ jest ono stosowane w standardzie.

Gdy obiekt jest inicjowany innym obiektem tego samego typu, nowy obiekt jest nazywany *kopią* obiektu inicjującego, nawet jeśli ta kopia została utworzona przy użyciu konstruktora przenoszącego. Niestety nie ma żadnej terminologii w C++, która rozróżniałaby między obiektem, który jest kopią skonstruowaną za pomocą kopiowania, a kopią skonstruowaną za pomocą przenoszenia:

```
void someFunc(Widget w);        // parametr w funkcji someFunc
                                // jest przekazywany przez wartość

Widget wid;                     // wid to pewien obiekt klasy Widget

someFunc(wid);                  // w tym wywołaniu funkcji someFunc,
                                // w jest kopią obiektu wid, która
                                // została utworzona za pomocą
                                // konstrukcji kopiującej

someFunc(std::move(wid));       // w tym wywołaniu funkcji someFunc,
                                // w jest kopią obiektu wid, która
```

```
                                    // zostałautworzona za pomocą
                                    // konstrukcji przenoszącej
```

Kopie r-wartości są zasadniczo konstruowane przez przenoszenie, a kopie l-wartości są zwykle konstruowane przez kopiowanie. Wynika z tego, że jeśli wiemy tylko, że obiekt jest kopią innego obiektu, nie możemy określić, jak kosztowne było skonstruowanie tej kopii. Na przykład w powyższym kodzie nie ma sposoby określenia, jak kosztowne jest utworzenie parametru `w` bez wiedzy, czy do funkcji `someFunc` zostały przekazane r-wartości, czy l-wartości. (Konieczna byłaby także znajomość kosztu przenoszenia i kopiowania obiektów `Widget`).

W wywołaniu funkcji wyrażenia przekazywane po stronie wywołującej są *argumentami* funkcji. Argumenty są używane do inicjowania *parametrów* funkcji. W pierwszym wywołaniu funkcji `someFunc` (powyżej) argumentem jest `wid`. W drugim wywołaniu argumentem jest `std::move(wid)`. W obu wywołaniach parametrem jest `w`. Rozróżnianie miedzy argumentami a parametrami jest ważne, ponieważ parametry są l-wartościami, ale argumenty, którymi są inicjowane, mogą być r-wartościami lub l-wartościami. Jest to szczególnie ważne w procesie *przekazywania doskonałego*, gdzie argument przekazywany do funkcji jest przekazywany do drugiej funkcji w taki sposób, że r-wartościowość lub l-wartościowość oryginalnego argumentu jest zachowywana. (Doskonałe przekazywanie jest szczegółowo opisane w punkcie 30).

Dobrze zaprojektowane funkcje są *bezpieczne wątkowo*, co oznacza, że oferują przynajmniej podstawową gwarancję bezpieczeństwa wątkowego (czyli *gwarancję podstawową*). Takie funkcje zapewniają w kodzie wywołującym, że nawet w przypadku wywołania wyjątku niezmienniki programu pozostają nietknięte (czyli żadne struktury danych nie zostaną uszkodzone), a żadne zasoby nie wycieką. Funkcje oferujące silną gwarancję bezpieczeństwa wątkowego (czyli *gwarancję silną*) zapewniają kod wywołujący, że w przypadku powstania wyjątku stan programu pozostaje taki jak przed wywołaniem.

Gdy odwołują się do *obiektu funkcyjnego*, zwykle mam na myśli obiekt typu obsługującego funkcję składową `operator()`. Innymi słowy, obiekt, który działa jak funkcja. Czasami używam tego terminu w nieco bardziej ogólnym sensie, aby określić cokolwiek, co może być wywołane przy użyciu składni wywołania funkcji globalnej (czyli „`functionName(arguments)`”). Ta szersza definicja obejmuje nie tylko obiekty obsługujące `operator()`, ale także funkcje i wskaźniki funkcji w stylu języka C. (Węższa definicja pochodzi z C++98, szersza z C++11). Dalsze uogólnienie przez dodanie wskaźników funkcji prowadzi do tego, co jest nazywane *obiektami wywoływalnymi*. Możemy zasadniczo zignorować dokładnie rozróżnienie i po prostu myśleć

o obiektach funkcyjnych i obiektach wywoływalnych jako elementach języka C++, które mogą być wywoływane przy użyciu pewnego rodzaju składni wywoływania funkcji.

Obiekty funkcyjne tworzone przez wyrażenia lambda są nazywane *domknięciami*. Rzadko konieczne jest rozróżnianie między wyrażeniami lambda a tworzonymi przez nie domknięciami, więc często będę określał jedno i drugie jako *wyrażenia lambda*. Analogicznie rzadko rozróżniam miedzy *szablonami funkcji* (czyli szablonami, które generują funkcje) a *funkcjami szablonowymi* (czyli funkcjami generowanymi na podstawie szablonów funkcji). Podobnie w przypadku *szablonów klas* i *klas szablonowych*.

Wiele rzeczy w języku C++ może być zarówno deklarowane, jak i definiowane. *Deklaracje* wprowadzają nazwy i typy bez podawania szczegółów, takich jak to, gdzie jest umieszczana pamięć lub jaka jest implementacja:

```
extern int x;                           // deklaracja obiektu

class Widget;                           // deklaracja klasy

bool func(const Widget& w);             // deklaracja funkcji

enum class Color;                       // deklaracja wyliczenia z zasięgiem
                                        // (patrz punkt 10)
```

Definicje dostarczają informacje o lokalizacji w pamięci lub implementacji:

```
int x;                                  // definicja obiektu

class Widget {                          // definicja klasy
  …
};

bool func(const Widget& w)
{ return w.size() < 10; }               // definicja funkcji

enum class Color
{ Yellow, Red, Blue };                  // definicja wyliczenia z zasięgiem
```

Definicja także zalicza się do deklaracji, dlatego, o ile nie jest naprawdę ważne, że coś jest definicją, zwykle odwołuję się do deklaracji.

Definiuję *sygnaturę* funkcji jako część jej deklaracji, która określa typy parametrów i wartości zwracanych. Nazwy funkcji i parametrów nie są częścią sygnatury. W powyższym przykładzie sygnaturą funkcji `func` jest `bool(const Widget&)`. Elementy deklaracji funkcji inne niż typy parametrów i wartości zwracanych (np. `noexcept` lub `constexpr`, jeśli są obecne) są wykluczane (`noexcept` i `constexpr` są opisane w punktach *14* i *15*). Oficjalna definicja „sygnatury" jest trochę inna od mojej, ale w tej książce moja definicja jest przydatna. (Oficjalna definicja czasami pomija typy zwracane).

Nowe standardy języka C++ zasadniczo zachowują poprawność kodu zapisanego w starszych standardach, ale czasami komitet standaryzacyjny określa funkcjonalności jako *przestarzałe (deprecate)*. Takie funkcjonalności oczekują w celi śmierci i zostaną usunięte z przyszłych standardów. Kompilatory mogą, ale nie muszą ostrzegać o używaniu przestarzałych funkcjonalności, jednak lepiej ich unikać. Nie tylko mogą prowadzić do późniejszych kłopotów z przenoszeniem, ale zasadniczo są gorsze od funkcjonalności, którymi zostały zastąpione. Na przykład wskaźnik `std::auto_ptr` jest przestarzały w C++11, ponieważ `std::unique_ptr` robi to samo, tylko lepiej.

Czasami standard mówi, że wynikiem operacji jest *niezdefiniowane działanie*. Oznacza to, że działanie w trybie wykonania jest nieprzewidywalne i powinno być oczywiste, że chcesz uniknąć takiej niepewności. Przykłady czynności o niezdefiniowanym działaniu obejmują użycie nawiasów kwadratowych („`[]`") do indeksowania poza granicami wektora `std::vector`, wyłuskiwanie niezainicjowanego iteratora lub spowodowanie wyścigu danych (czyli występowanie co najmniej dwóch wątków, z których przynajmniej jeden zapisuje, równocześnie uzyskujących dostęp do tej samej lokalizacji w pamięci).

Wbudowane wskaźniki, takie jak zwracane przez instrukcję `new`, nazywam *wskaźnikami surowymi*. Przeciwieństwem wskaźnika surowego jest *wskaźnik inteligentny*. Wskaźniki inteligentne zwykle przeciążają operatory wyłuskiwania wskaźników (`operator->` i `operator*`), chociaż, jak wyjaśnię w punkcie 20, wskaźnik `std::weak_ptr` jest wyjątkiem.

Raportowanie błędów i sugerowanie poprawek

Dołożyłem wszelkich starań, aby wypełnić tę książkę jasnymi, dokładnymi i przydatnymi informacjami, ale na pewno można ją jeszcze poprawić. Jeśli znajdziesz błędy dowolnego typu (techniczne, logiczne, gramatyczne, typograficzne itp.) lub jeśli masz sugestie dotyczące sposobów udoskonalenia tej książki, napisz do mnie na adres *emc++@aristeia.com*. Nowe dodruki dają mi możliwość zmodyfikowania książki *Skuteczny nowoczesny C++*, a nie mogę rozwiązać problemów, o których nie wiem!

Aby wyświetlić listę problemów, o których już wiem, zajrzyj na stronę erraty książki pod adresem *http://www.aristeia.com/BookErrata/emc++-errata.html*.

Rozdział 1

Dedukcja typów

W języku C++98 istniał jeden zestaw reguł dedukcji typów używany w szablonach funkcji. W standardzie C++11 zmodyfikowano go odrobinę i dodano dwa kolejne zestawy reguł, które dotyczą deklaracji `auto` i instrukcji `decltype`. W standardzie C++14 konteksty użycia technik `auto` i `decltype` zostały rozszerzone. Dzięki coraz obszerniejszym zastosowaniom dedukcji typów nie musimy pisać ich nazw wielokrotnie ani w oczywistych sytuacjach. Powoduje to, że oprogramowanie C++ lepiej się adaptuje, ponieważ zmiana typu w jednym miejscu kodu źródłowego automatycznie propaguje się za pomocą dedukcji typów do innych lokalizacji. Jednak utworzony kod może być trudniejszy do zrozumienia, ponieważ typy dedukowane przez kompilator mogą nie być tak oczywiste, jak tego oczekujemy.

Bez solidnej wiedzy na temat sposobu działania dedukcji typów skuteczne programowanie w nowoczesnym języku C++ jest prawie niemożliwe. Po prostu technika ta występuje w zbyt wielu kontekstach: w wywołaniach szablonów funkcji, w większości sytuacji, gdzie występuje deklaracja `auto`, w wyrażeniach `decltype`, a w standardzie C++14 również tam, gdzie jest używana enigmatyczna konstrukcja `decltype(auto)`.

W tym rozdziale zawarte zostały informacje na temat dedukcji typów potrzebne każdemu programiście C++. Zawiera on objaśnienie dedukcji typów w szablonach, opartego na niej działania deklaracji `auto`, a także odmiennego sposobu działania instrukcji `decltype`. Dowiemy się także, jak zmusić kompilatory do uwidocznienia wyników dedukcji typów, abyśmy mogli upewnić się, że odbywa się ona zgodnie z oczekiwaniami.

Punkt 1: Dedukcja typów w szablonach

Kiedy użytkownicy złożonego systemu nie mają pojęcia, jak on działa, ale są z niego zadowoleni, mówi to wiele o projekcie systemu. Pod tym względem dedukcja typów w szablonach w języku C++ jest ogromnym sukcesem. Miliony programistów

przekazują argumenty do funkcji szablonowych z całkowicie satysfakcjonującymi wynikami, chociaż większość z nich czułaby się przyciśnięta do muru, gdyby mieli dać więcej niż mglisty opis sposobu dedukcji typów używanych przez te funkcje.

Jeśli należysz do tej grupy, mam dobre i złe wiadomości. Dobra wiadomość to ta, że dedukcja typów dla szablonów jest podstawą jednej z najbardziej interesujących funkcjonalności nowoczesnego języka C++: `auto`. Osoby zadowolone z tego, jak zachodziła dedukcja typów w C++98 dla szablonów, będą na pewno zadowolone ze sposobu dedukcji typów C++11 dla deklaracji `auto`. Zła wiadomość jest taka, że gdy reguły dedukcji typów w szablonach są stosowane w kontekście `auto`, czasami działają mniej intuicyjnie niż przy stosowaniu ich do szablonów. Dlatego ważne jest, aby dobrze zrozumieć aspekty dedukcji typów w szablonach, na których oparta jest deklaracja `auto`. Wszystkie potrzebne informacje zawarte są w tym Sposobie.

Jeśli zechcemy przyjrzeć się fragmentowi pseudokodu, szablon funkcji możemy przedstawić tak:

```
template<typename T>
void f(ParamType param);
```

Wywołanie może wyglądać tak:

```
f(expr);                    // wywołanie funkcji f z pewnym wyrażeniem
```

Podczas kompilacji kompilatory używają wyrażenia *expr* do dedukcji dwóch typów: jednego dla `T`, a drugiego dla *ParamType*. Te typy są często różne, ponieważ *ParamType* często zawiera dodatkowe określenia, np. `const` lub kwalifikatory odwołań. Jeśli na przykład szablon jest zadeklarowany w taki sposób

```
template<typename T>
void f(const T& param);     // ParamType to const T&
```

i mamy takie wywołanie

```
int x = 0;

f(x);                       // wywołanie funkcji f z argumentem int
```

wydedukowany typ `T` to `int`, ale wydedukowany typ *ParamType* to `const int&`.

Oczekiwanie, że wydedukowany typ `T` jest taki sam jak typ argumentu przekazanego do funkcji (czyli że `T` ma typ wyrażenia *expr*), jest naturalne. W powyższym

przykładzie tak jest: x ma typ int i T ma wydedukowany typ int. Ale to nie zawsze działa w ten sposób. Wydedukowany typ T zależy nie tylko od typu wyrażenia *expr*, ale także od formy *ParamType*. Istnieją tu trzy przypadki:

- *ParamType* to wskaźnik lub typ referencyjny, ale nie jest odwołaniem uniwersalnym. (Odwołania uniwersalne są opisane w punkcie 24. Teraz wystarczy wiedzieć, że istnieją i różnią się od odwołań do l-wartości i odwołań do r-wartości).
- *ParamType* to odwołanie uniwersalne.
- *ParamType* nie jest ani wskaźnikiem, ani odwołaniem.

Dlatego mamy do zbadania trzy przypadki dedukcji typów. Każdy z nich będzie oparty na ogólnej formie szablonów i ich wywołań:

```
template<typename T>
void f(ParamType param);

f(expr);                // dedukcja T i ParamType na podstawie expr
```

Przypadek 1: *ParamType* to odwołanie lub wskaźnik, ale nie odwołanie uniwersalne

Najprostsza sytuacja występuje, gdy *ParamType* ma typ referencyjny lub wskaźnikowy, ale nie jest odwołaniem uniwersalnym. W takim przypadku dedukcja typu działa następująco:

1. Jeśli typ wyrażenia *expr* to odwołanie, zignoruj jego referencyjność.
2. Następnie dopasuj do wzorca typ wyrażenia *expr* względem *ParamType*, aby wyznaczyć T.

Jeśli weźmiemy taki przykład

```
template<typename T>
void f(T& param);       // param to odwołanie
```

i takie deklaracje danych

```
int x = 27;             // x to int
const int cx = x;       // cx to const int
const int& rx = x;      // rx to odwołanie do x jako const int
```

wydedukowane typy param i T w różnych wywołaniach będą następujące:

```
f(x);                   // T to int, typ param to int&

f(cx);                  // T to const int,
                        // typ param to const int&
f(rx);                  // T to const int,
                        // to param to const int&
```

W drugim i trzecim wywołaniu można zauważyć, że ponieważ `cx` i `rx` oznaczają wartości `const`, typ `T` jest dedukowany jako `const int`, co powoduje, że typem parametru jest `const int&`. Jest to ważne podczas wywoływania. Gdy przekazujemy obiekt `const` do parametru odwołania, oczekujemy, że obiekt pozostanie niezmodyfikowany, czyli parametr będzie odwołaniem do stałej (`const`). Dlatego przekazanie obiektu `const` do szablonu przyjmującego parametr `T&` jest bezpieczne: stałość (`const`) obiektu staje się częścią wydedukowanego typu `T`.

W trzecim przykładzie wprawdzie typ `rx` to odwołanie, jednak typ `T` jest dedukowany jako niereferencyjny. Dzieje się tak, ponieważ referencyjność `rx` jest ignorowana podczas dedukcji typu.

Wszystkie te przykłady pokazały parametry odwołań do l-wartości, ale dedukcja typów działa dokładnie tak samo w przypadku parametrów odwołań do r-wartości. Oczywiście tylko argumenty r-wartości mogą być przekazywane przez parametry odwołań do r-wartości, ale to ograniczenie nie ma nic wspólnego z dedukcją typów.

Jeśli zmienimy typ parametru funkcji `f` z `T&` na `const T&`, działanie zmieni się odrobinę, ale w niezbyt zaskakujący sposób. Stałość (`const`) danych `cx` i `rx` nadal będzie uwzględniana, ale ponieważ teraz zakładamy, że `param` jest odwołaniem do stałej (`const`), nie ma już potrzeby dedukcji deklaracji `const` jako części typu `T`:

```
template<typename T>
void f(const T& param);  // param to teraz odwołanie do const

int x = 27;              // jak poprzednio
const int cx = x;        // jak poprzednio
const int& rx = x;       // jak poprzednio

f(x);                    // T to int, typ param to const int&

f(cx);                   // T to int, typ param to const int&
```

```
f(rx);                  // T to int, typ param to const int&
```

Jak poprzednio, referencyjność `rx` jest ignorowana podczas dedukcji typu.

Jeżeli argument `param` byłby wskaźnikiem (lub wskaźnikiem do stałej, czyli `const`), zamiast odwołaniem, działanie byłoby dokładnie takie samo:

```
template<typename T>
void f(T* param);       // param to teraz wskaźnik

int x = 27;             // jak poprzednio
const int *px = &x;     // px to wskaźnik do x jako const int


f(&x);                  // T to int, typ param to int*

f(px);                  // T to const int,
                        // typ param to const int*
```

Na pewno już ziewasz i przysypiasz, ponieważ reguły dedukcji typów w języku C++ działają tak naturalnie dla parametrów będących odwołaniami i wskaźnikami, że widziane w formie pisemnej są naprawdę nudne. Wszystko jest tak oczywiste! Tego właśnie oczekujemy w systemie dedukcji typów.

Przypadek 2: *ParamType* jest odwołaniem uniwersalnym

Mniej oczywiste jest działanie szablonów przyjmujących parametry odwołań uniwersalnych. Takie parametry są deklarowane jako odwołania do r-wartości (np. w szablonie funkcji przyjmującym parametr `T`, typem deklarowanym odwołania uniwersalnego jest `T&&`), ale działają inaczej, gdy przekażemy do nich argumenty będące l-wartościami. Dokładny opis znajduje się w punkcie 24, a jego wersja skrócona jest taka:

- Jeżeli *expr* to l-wartość, zarówno `T`, jak i *ParamType* są dedukowane jako odwołania do l-wartości. Jest to podwójnie niezwykłe. Po pierwsze jest to jedyna sytuacja dotycząca dedukcji typów szablonu, gdzie `T` jest dedukowane jako odwołanie. Po drugie, chociaż *ParamType* jest deklarowany przy użyciu składni dla odwołania do r-wartości, jego dedukowanym typem jest odwołanie do l-wartości.
- Jeżeli *expr* to r-wartość, stosowane są „normalne” reguły (czyli Przypadek 1).

Na przykład:

```
template<typename T>
void f(T&& param);       // param jest teraz odwołaniem uniwersalnym

int x = 27;              // jak poprzednio
const int cx = x;        // jak poprzednio
const int& rx = x;       // jak poprzednio

f(x);                    // x to l-wartość, więc T to int&,
                         // typ param to również int&

f(cx);                   // cx to l-wartość, więc T to const int&,
                         // typ param to także const int&

f(rx);                   // rx to l-wartość, więc T to const int&,
                         // typ param to także const int&

f(27);                   // 27 to r-wartość, więc T to int,
                         // dlatego typ param to int&&
```

W punkcie 24 zostało wyjaśnione dokładnie, dlaczego te przykłady tak działają. Najistotniejsze tutaj jest to, że reguły dedukcji typów dla parametrów odwołań uniwersalnych są inne od tych dla parametrów, które są odwołaniami do l-wartości lub odwołaniami do r-wartości. W szczególności, gdy są używane odwołania uniwersalne, dedukcja typów rozróżnia argumenty l-wartości od argumentów r-wartości. Nie dzieje się tak nigdy w przypadku odwołań innych niż uniwersalne.

Przypadek 3: *ParamType* nie jest ani wskaźnikiem, ani odwołaniem

Gdy *ParamType* nie jest ani wskaźnikiem, ani odwołaniem, mamy do czynienia z przekazywaniem przez wartość:

```
template<typename T>
void f(T param);         // param jest teraz przekazywany przez wartość
```

Oznacza to, że `param` będzie kopią tego, co zostanie przekazane – całkowicie nowym obiektem. Fakt, że `param` będzie nowym obiektem, wpływa na reguły rządzące sposobem dedukcji `T` na podstawie wyrażenia *expr*:

1. Jak poprzednio, jeśli typ *expr* jest odwołaniem, ignorujemy część referencyjną.
2. Jeżeli po zignorowaniu referencyjności w wyrażeniu *expr*, wyrażenie *expr* ma właściwość `const`, ignorujemy również ten fakt. Jeżeli jest obiektem `volatile`, również to ignorujemy. (Obiekty `volatile` są bardzo rzadko spotykane. Zasadniczo wykorzystuje się je tylko przy implementacji sterowników urządzeń. Szczegółowy opis znajdziesz w punkcie 40).

Stąd:

```
int x = 27;           // jak poprzednio
const int cx = x;     // jak poprzednio
const int& rx = x;    // jak poprzednio

f(x);                 // oba typy T i param to int

f(cx);                // ponownie oba typy T i param to int

f(rx);                // nadal oba typy T i param to int
```

Zauważmy, że chociaż `cx` i `rx` reprezentują wartości `const`, `param` nie ma właściwości `const`. Ma to sens. `param` jest obiektem całkowicie niezależnym od `cx` i `rx` – *kopią* `cx` lub `rx`. Fakt, że `cx` i `rx` nie mogą być modyfikowane, nie ma wpływu na to, czy można modyfikować `param`. Dlatego określenia `const` (i ewentualnie `volatile`) wyrażenia *expr* są ignorowane podczas dedukcji typu parametru `param`: po prostu dlatego, że niemożność modyfikacji wyrażenia *expr* nie oznacza, że nie można modyfikować jego kopii.

Ważne jest, aby zauważyć, że określenie `const` (i `volatile`) jest ignorowane tylko w przypadku parametrów przekazywanych przez wartość. Jak zobaczyliśmy, w przypadku parametrów, które są odwołaniami lub wskaźnikami do stałych (`const`), właściwość `const` wyrażenia *expr* jest zachowywana. Rozważmy jednak przypadek, gdy *expr* jest stałym (`const`) wskaźnikiem do stałego (`const`) obiektu, a wyrażenie *expr* jest przekazywane do parametru `param` przez wartość:

```
template<typename T>
void f(T param);         // param jest nadal przekazywany przez wartość

const char* const ptr =  // ptr to stały wskaźnik do stałego obiektu
  "Fun with pointers";
```

```
f(ptr);                  // przekazanie argumentu typu const char * const
```

Tutaj const po prawej stronie gwiazdki powoduje deklarację wskaźnika ptr jako const: ptr nie może wskazać na inną lokalizację, ani nie może mieć przypisanej wartości null. (const po lewej stronie od gwiazdki oznacza, że to, na co wskazuje wskaźnik ptr – ciąg znaków – jest również const, więc nie może być zmodyfikowany). Gdy argument ptr jest przekazywany do funkcji f, bity stanowiące wskaźnik są kopiowane do parametru param. W ten sposób *sam wskaźnik (ptr) jest przekazywany przez wartość*. W połączeniu z regułą dedukcji typów dla parametrów przekazywanych przez wartość, właściwość const wskaźnika ptr zostanie zignorowana, a wydedukowanym typem parametru param będzie const char*, czyli modyfikowalny wskaźnik do stałego (const) ciągu znaków. Właściwość const tego, na co wskazuje wskaźnik ptr, jest zachowywana podczas dedukcji typu, ale właściwość const samego wskaźnika ptr jest ignorowana podczas kopiowania w celu utworzenia nowego wskaźnika param.

Argumenty tablicowe

Mniej więcej opisałem główne zasady dedukcji typów szablonowych, ale istnieje niszowy przypadek, o którym warto wiedzieć. Chodzi o to, że typy tablicowe różnią się od typów wskaźnikowych, chociaż czasami wydaje się, że można je wymieniać. Główną przyczyną tej iluzji jest to, że w wielu kontekstach tablicę można *sprowadzić* do wskaźnika na jej pierwszy element. To sprowadzenie sprawia, że da się skompilować taki kod, jak poniższy:

```
const char name[] = "J. P. Briggs";  // typ name to
                                     // const char[13]

const char * ptrToName = name;       // tablica jest sprowadzana
                                     // do wskaźnika
```

Tutaj wskaźnik const char* o nazwie ptrToName jest inicjowany za pomocą zmiennej name typu const char[13]. Te typy (const char* i const char[13]) nie są takie same, ale z powodu reguły sprowadzania tablicy do wskaźnika kod się kompiluje.

Co jednak stanie się wtedy, kiedy przekażemy tablicę do szablonu przyjmującego parametr przez wartość?

```
template<typename T>
void f(T param);      // szablon z parametrem przez wartość
```

```
f(name);              // jakie typy są dedukowane dla T i param?
```

Zaczniemy od spostrzeżenia, że nie ma czegoś takiego jak parametr funkcji, który jest tablicą. Tak, tak, poniższa składnia jest dozwolona:

```
void myFunc(int param[]);
```

ale deklaracja funkcji jest traktowana jako deklaracja wskaźnika, co oznacza, że funkcja `myFunc` mogłaby być równoważnie zadeklarowana tak:

```
void myFunc(int* param);         // ta sama funkcja, co wyżej
```

Ta równoważność parametrów tablicowych i wskaźnikowych wynika z zakorzenienia języka C++ w języku C i powoduje mylne wrażenie, że typy tablicowe i wskaźnikowe są takie same.

Ponieważ deklaracje parametrów tablicowych są traktowane tak, jak gdyby były parametrami wskaźnikowymi, typ tablicy przekazywany do funkcji przez wartość jest dedukowany jako typ wskaźnikowy. Oznacza to, że w wywołaniu szablonu `f` jego parametr typu `T` jest dedukowany jako `const char*`:

```
f(name);       // name to tablica, ale T jest dedukowane jako const char*
```

Ale w tym tkwi haczyk. W funkcjach nie można deklarować parametrów, które są prawdziwymi tablicami, natomiast *można* w nich deklarować parametry, które są *odwołaniami* do tabel! Dlatego, jeżeli zmodyfikujemy szablon `f`, aby przyjmował argument przez odwołanie

```
template<typename T>
void f(T& param);      // szablon z parametrem przez odwołanie
```

i przekażemy do niego tablicę

```
f(name);               // przekazanie tablicy do f
```

typem wydedukowanym dla `T` jest faktyczny typ tablicy! Ten typ zawiera rozmiar tablicy, więc w tym przykładzie `T` jest dedukowany jako `const char [13]`, a typem parametru funkcji `f` (odwołania do tablicy) jest `const char (&)[13]`. Tak, składnia wygląda toksycznie, ale jej znajomość pozwala zapunktować u tych nielicznych osób, które o to dbają.

Interesujące jest oto, że możliwość deklarowania odwołań do tablic umożliwia tworzenie szablonu służącego do dedukcji liczby elementów zawartych w tablicy:

```
// zwraca rozmiar tablicy jako stałą czasu kompilacji.
// (Parametr tablicowy nie ma nazwy, ponieważ interesuje nas
// tylko liczba zawartych w tablicy elementów).
template<typename T, std::size_t N>                // zobacz poniższe
constexpr std::size_t  arraySize(T (&)[N]) noexcept // informacje
{                                                  // o constexpr
  return N;                                        // i noexcept
}
```

Zgodnie z opisem zawartym w punkcie 15 zadeklarowanie tej funkcji jako `constexpr` sprawia, że jej wynik jest dostępny w czasie kompilacji. Dlatego możliwe jest zadeklarowanie, powiedzmy, tablicy z taką samą liczbą elementów, jaką ma inna tablica, której rozmiar jest obliczony na podstawie inicjatora klamrowego:

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };      // keyVals ma
                                                 // 7 elementów

int mappedVals[arraySize(keyVals)];              // tyle samo ma
                                                 // mappedVals
```

Oczywiście, jako nowoczesny programista C++, wolisz typ `std::array` od wbudowanego typu tablicowego:

```
std::array<int, arraySize(keyVals)> mappedVals;  // rozmiar mappedVals
                                                 // wynosi 7
```

Zadeklarowanie `arraySize` jako `noexcept` pomaga kompilatorom generować lepszy kod. Szczegółowe informacje znajdziesz w punkcie 14.

Argumenty funkcyjne

Tablice nie są jedynymi elementami języka C++, które można sprowadzić do wskaźników. Typy funkcji można sprowadzić do wskaźników funkcji, a wszystko, co omówiliśmy odnośnie dedukcji typów dla tablic, dotyczy dedukcji typów dla funkcji oraz ich sprowadzania do wskaźników funkcji. W efekcie otrzymujemy:

```
void someFunc(int, double);   // someFunc to funkcja;
```

```
                                // jej typ to void(int, double)

template<typename T>
void f1(T param);     // w f1 param jest przekazywany przez wartość

template<typename T>
void f2(T& param);    // w f2 param jest przekazywany przez odwołanie

f1(someFunc);         // param jest dedukowany jako wskaźnik do funkcji;
                      // typ to void (*)(int, double)
f2(someFunc);         // param jest dedukowany jako odwołanie do funkcji;
                      // typ to void (&)(int, double)
```

W praktyce różnica jest prawie nieistotna, ale jeżeli chcesz wiedzieć o sprowadzaniu tablicy do wskaźnika, możesz także dowiedzieć się o sprowadzaniu funkcji do wskaźnika.

I tu dochodzimy do sedna: reguł związanych z deklaracją `auto` dotyczących dedukcji typów w szablonach. Wspomniałem na początku, że są one dość proste, i przeważnie takie są. Specjalne traktowanie uzgodnionych l-wartości podczas dedukcji typów dla odwołań uniwersalnych wprowadza odrobinę zamieszania, a reguły sprowadzania do wskaźnika dotyczące tablic i funkcji wprowadzają jeszcze większy zamęt. Czasami po prostu chcemy zażądać od kompilatora „Powiedz mi, jaki typ dedukujesz!" Wtedy warto skorzystać z punktu 4, w którym pokażę, jak skłonić kompilatory do takiego działania.

Warto zapamiętać

- Podczas dedukcji typu w szablonie argumenty będące odwołaniami nie są traktowane jako odwołania, czyli ich referencyjność jest ignorowana.
- Podczas dedukcji typów dla parametrów odwołań uniwersalnych argumenty l-wartości są traktowane w specjalny sposób.
- Podczas dedukcji typów dla parametrów przekazywanych przez wartość argumenty `const` i/lub `volatile` są traktowane jako pozbawione tych cech.
- Podczas dedukcji typu w szablonie argumenty, które są nazwami tablic lub funkcji, są sprowadzane do wskaźników, o ile nie użyjemy ich do inicjowania odwołań.

Punkt 2: Dedukcja typu auto

Po przeczytaniu punktu 1 dotyczącego dedukcji typów w szablonach, wiesz już prawie wszystko, co trzeba wiedzieć na temat dedukcji typów `auto`, ponieważ poza tylko jednym osobliwym wyjątkiem dedukcja typów `auto` *to jest* dedukcja typów w szablonach. Jak to możliwe? Dedukcja typów w szablonach obejmuje szablony, funkcje i parametry, natomiast `auto` nie ma z nimi nic do czynienia.

To prawda, ale nie ma znaczenia. Istnieje bezpośrednie odwzorowanie dedukcji typów w szablonach na dedukcję typów `auto`. Istnieje dokładna algorytmiczna transformacja jednego rodzaju w drugi.

W punkcie 1 dedukcja typów w szablonach została wyjaśniona na przykładzie tego ogólnego szablonu funkcji:

```
template<typename T>
void f(ParamType param);
```

i tego ogólnego wywołania:

```
f(expr);                    // wywołanie f z pewnym wyrażeniem
```

W wywołaniu funkcji `f` kompilatory korzystają z wyrażenia *`expr`* do dedukcji typów dla `T` i *`ParamType`*.

Gdy zmienne są deklarowane przy użyciu `auto`, `auto` odgrywa rolę `T` w szablonie, a określenie typu dla zmiennej pełni rolę *`ParamType`*. Jest to łatwiej pokazać niż opisać, więc rozważmy ten przykład:

```
auto x = 27;
```

Tutaj określeniem typu zmiennej `x` jest po prostu samo `auto`. Z drugiej strony w tej deklaracji

```
const auto cx = x;
```

określeniem typu jest `const auto`. Natomiast tutaj

```
const auto& rx = x;
```

określeniem typu jest `const auto&`. W celu dedukcji typów dla `x`, `cx` i `rx` w tych przykładach kompilatory działają tak, jakby dla każdej deklaracji był podany szablon, a także wywołanie do tego szablonu z odpowiednim wyrażeniem inicjującym:

```
template<typename T>                // szablon koncepcyjny do
void func_for_x(T param);           // dedukcji typu x

func_for_x(27);                     // wywołanie koncepcyjne: dedukowany
                                    // typ param to typ x

template<typename T>                // szablon koncepcyjny do
void func_for_cx(const T param);    // dedukcji typu cx

func_for_cx(x);                     // szablon koncepcyjny: dedukowany
                                    // typ param to typ cx

template<typename T>                // szablon koncepcyjny do
void func_for_rx(const T& param);   // dedukcji typu rx

func_for_rx(x);                     // szablon koncepcyjny: dedukowany
                                    // typ param to typ rx
```

Jak powiedziałem, dedukcja typu dla `auto` jest, z tylko jednym wyjątkiem (omówimy go wkrótce), taka sama jak dedukcja typów dla szablonów.

W punkcie 1 dedukcja typów w szablonach została podzielona na trzy przypadki w oparciu o charakterystykę *`ParamType`*, czyli określenia parametru `param` w ogólnym szablonie funkcji. W deklaracji zmiennej przy użyciu `auto`, określenie typu zajmuje miejsce *`ParamType`*, więc również istnieją trzy przypadki:

- Przypadek 1: Określeniem typu jest wskaźnik lub odwołanie, ale nie odwołanie uniwersalne.
- Przypadek 2: Określeniem typu jest odwołanie uniwersalne.
- Przypadek 3: Określeniem typu nie jest ani wskaźnik, ani odwołanie.

Widzieliśmy już przykłady przypadków 1 i 3:

```
auto x = 27;          // przypadek 3 (x nie jest wskaźnikiem ani
                      // odwołaniem)
```

```
const auto cx = x;     // przypadek 3 (cx również nie jest wskaźnikiem ani
                       // odwołaniem)

const auto& rx = x;    // przypadek 1 (rx jest odwołaniem
                       // nieuniwersalnym).
```

Przypadek 2 działa zgodnie z oczekiwaniami:

```
auto&& uref1 = x;      // x to l-wartość typu int,
                       // więc typ uref1 to int&

auto&& uref2 = cx;     // cx to l-wartość typu const int,
                       // więc typ uref2 to const int&

auto&& uref3 = 27;     // 27 to r-wartość typu int,
                       // więc typ uref3 to int&&
```

Punkt 1 kończy się omówieniem sprowadzania nazw tablic i funkcji do wskaźników na niereferencyjne określenia typów. Dzieje się tak również podczas dedukcji typów `auto`:

```
const char name[] =            // typ name to const char[13]
  "R. N. Briggs";

auto arr1 = name;              // typ arr1 to const char*

auto& arr2 = name;             // typ arr2 to
                               // const char (&)[13]

void someFunc(int, double);    // someFunc to funkcja;
                               // jej typ to void(int, double)

auto func1 = someFunc;         // typ func1 to
                               // void (*)(int, double)

auto& func2 = someFunc;        // typ func2 to
                               // void (&)(int, double)
```

Jak możemy zauważyć, dedukcja typów `auto` działa tak jak dedukcja typów w szablonach. Są to zasadniczo dwie strony tej samej monety.

Z jednym wyjątkiem. Zaczęliśmy od obserwacji, że jeżeli chcemy zadeklarować zmienną `int` z początkową wartością 27, składnia języka C++98 daje dwie możliwości:

```
int x1 = 27;
int x2(27);
```

Standard C++11, dzięki obsłudze inicjowania ujednoliconego, dodaje następujące opcje:

```
int x3 = { 27 };
int x4{ 27 };
```

Wszystko jedno, cztery składnie, ale tylko jeden wynik: zmienna `int` o wartości 27.

Jednak zgodnie z opisem w punkcie 5 deklaracja zmiennych przy użyciu `auto` zamiast stałych typów ma pewne zalety, więc miło byłoby zastąpić `int` słowem kluczowym `auto` w powyższych deklaracjach zmiennych. Proste tekstowe zastąpienie daje następujący kod:

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

Wszystkie te deklaracje się kompilują, ale ich znaczenie jest różne od znaczenia deklaracji przedstawionych poprzednio. Pierwsze dwie instrukcje faktycznie deklarują zmienną typu `int` o wartości 27. Jednak kolejne dwie deklarują zmienną typu `std::initializer_list<int>` zawierającą jeden element o wartości 27!

```
auto x1 = 27;             // typ to int, wartość to 27

auto x2(27);              // jw.

auto x3 = { 27 };         // typ to std::initializer_list<int>,
                          // wartość to { 27 }

auto x4{ 27 };            // jw.
```

Wynika to ze specjalnej reguły dedukcji typu dla `auto`. Gdy inicjator zmiennej deklarowanej jako `auto` jest umieszczony w nawiasach klamrowych, dedukowanym

typem jest std::initializer_list. Jeżeli taki typ nie może zostać wydedukowany (np. dlatego, że wartości w inicjatorze klamrowym są różnych typów), kod zostanie odrzucony:

```
auto x5 = { 1, 2, 3.0 };  // błąd! nie można wydedukować T dla
                          // std::initializer_list<T>
```

Zgodnie z komentarzem dedukcja typu w tym przypadku się nie powiedzie, ale ważne, aby rozpoznać, że występują tu faktycznie dwa rodzaje dedukcji typów. Jeden rodzaj wynika z użycia auto: trzeba wydedukować typ zmiennej x5. Ponieważ inicjator zmiennej x5 jest w nawiasach klamrowych, typ x5 musi zostać wydedukowany jako std::initializer_list. Jednak std::initializer_list to szablon. Instancjami są std::initializer_list<T> dla pewnego typu T, a to oznacza, że trzeba także wydedukować typ T. Taka dedukcja podlega pod występujący tutaj drugi typ dedukcji typów: dedukcję typów w szablonach. W tym przykładzie ta dedukcja się nie powiedzie, ponieważ wartości w inicjatorze klamrowym nie mają tego samego typu.

Traktowanie inicjatorów klamrowych jest jedyną różnicą w sposobach dedukcji typów auto i w dedukcji typów w szablonach. Gdy zmienna zadeklarowana jako auto jest inicjowana za pomocą inicjatora klamrowego, dedukowany typ jest instancją std::initializer_list. Jednak takie samo zainicjowanie odpowiedniego szablonu sprawi, że dedukcja kodu się nie powiedzie, a kod zostanie odrzucony:

```
auto x = { 11, 23, 9 };   // typ x tp
                          // std::initializer_list<int>

template<typename T>      // szablon z parametrem
void f(T param);          // deklaracja równoważna do
                          // deklaracji x

f({ 11, 23, 9 });         // błąd! nie można wydedukować typu T
```

Jeśli jednak określimy szablon, gdzie param jest typu std::initializer_list<T> dla pewnego nieznanego typu T, dedukcja typu T w szablonie da następujący wynik:

```
template<typename T>
void f(std::initializer_list<T> initList);

f({ 11, 23, 9 });         // T wydedukowany jako int, a typ initList
                          // to std::initializer_list<int>
```

Dlatego jedyną prawdziwą różnicą między dedukcją typów auto a dedukcją typów w szablonach jest to, że dedukcja auto *zakłada*, że inicjator klamrowy reprezentuje std::initializer_list, a dedukcja typów w szablonie – nie.

Możesz zastanawiać się, dlaczego dedukcja typów auto ma specjalną regułę dla inicjatorów klamrowych, a dedukcja typów w szablonie – nie. Też nad tym myślałem. Niestety nie udało mi się znaleźć żadnego przekonującego wyjaśnienia. Jednak reguła jest regułą, a to oznacza, że musimy zapamiętać, że jeśli deklarujemy zmienną przy użyciu auto i inicjujemy ją przy użyciu inicjatora klamrowego, wydedukowanym typem zawsze będzie std::initializer_list. Szczególnie ważne, aby mieć to na uwadze, jeżeli przyjmiemy filozofię inicjowania ujednoliconego – rutynowe umieszczanie początkowych wartości w klamrach. Klasycznym błędem w programowaniu C++11 jest przypadkowe zadeklarowanie zmiennej std::initializer_list, gdy chcemy zadeklarować coś innego. Ta pułapka jest jednym z powodów, dla których niektórzy programiści umieszczają nawiasy klamrowe wokół inicjatorów tylko wtedy, gdy muszą. (Kiedy jest to koniecznie, zostało opisane w punkcie 7).

To już wszystko na ten temat w języku C++11, ale standard C++14 niesie dalszy ciąg. Język C++14 dopuszcza, aby słowo kluczowe auto wskazywało, że typ zwracany przez funkcję powinien zostać wydedukowany (patrz punkt 3), a także dopuszcza używanie deklaracji auto w parametrach wyrażeń lambda. Jednak te zastosowania auto korzystają z *dedukcji typów w szablonach*, a nie dedukcji typu auto. Dlatego funkcja zwracająca typ auto podany jako inicjator klamrowy nie da się skompilować:

```
auto createInitList()
{
  return { 1, 2, 3 };          // błąd: nie można wydedukować typu
}                              // dla { 1, 2, 3 }
```

To samo dotyczy użycia auto w specyfikacji typu parametru w wyrażeniu lambda w języku C++14:

```
std::vector<int> v;
…
auto resetV =
  [&v](const auto& newValue) { v = newValue; };     // C++14
…

resetV({ 1, 2, 3 });           // błąd! nie można wydedukować typu
                               // dla { 1, 2, 3 }
```

Warto zapamiętać

- Dedukcja typów `auto` zwykle przebiega tak samo jak dedukcja typów w szablonach, ale w dedukcji typów `auto` założono, że inicjator klamrowy reprezentuje `std::initializer_list`, inaczej niż w dedukcji typów w szablonach.
- Słowo kluczowe `auto` określające typ zwracany przez funkcję lub parametr w wyrażeniu lambda powoduje dedukcję typów w szablonie, a nie dedukcję typów `auto`.

Punkt 3: `decltype`

Instrukcja `decltype` to osobliwy twór. Gdy podamy nazwę lub wyrażenie, instrukcja `decltype` poinformuje o typie tej nazwy lub wyrażenia. Zazwyczaj ta informacja jest dokładnie zgodna z przewidywaniami. Czasami jednak prowadzi do wyników, które sprawiają, że zaczynamy drapać się po głowie i szukać objaśnienia w opracowaniach lub witrynach z pytaniami i odpowiedziami.

Zaczniemy od typowych przypadków – tych, które nie skrywają żadnych niespodzianek. W przeciwieństwie do tego, co dzieje się podczas dedukcji typów w szablonach i `auto` (patrz punkty 1 i 2), instrukcje `decltype` zwykle powtarzają dokładnie typ nazwy lub wyrażenia, które podamy:

```
const int i = 0;            // decltype(i) to const int

bool f(const Widget& w);    // decltype(w) to const Widget&
                            // decltype(f) to bool(const Widget&)

struct Point {
  int x, y;                 // decltype(Point::x) to int
};                          // decltype(Point::y) to int

Widget w;                   // decltype(w) to Widget

if (f(w)) …                 // decltype(f(w)) to bool

template<typename T>        // uproszczona wersja std::vector
```

```
class vector {
public:
  …
  T& operator[](std::size_t index);
  …
};

vector<int> v;              // decltype(v) to vector<int>
…
if (v[0] == 0) …            // decltype(v[0]) to int&
```

Widzisz? Żadnych niespodzianek.

W języku C++11 przypuszczalnie głównym zastosowaniem instrukcji `decltype` jest deklaracja szablonów funkcji, w których typ zwracany przez funkcję zależy od typów jej parametrów. Na przykład załóżmy, że chcemy napisać funkcję, która przyjmuje kontener obsługujący indeksowanie za pomocą nawiasów kwadratowych (czyli „`[]`”) oraz indeksu, a ponadto uwierzytelnia użytkownika przed zwróceniem wyników operacji indeksowania. Typem zwracanym przez funkcję powinien być taki sam typ, jak zwracany przez operację indeksowania.

Operator `operator[]` użyty na kontenerze obiektów typu `T` zwykle zwraca `T&`. Jest tak np. w przypadku `std::deque` i jest tak prawie zawsze w przypadku `std::vector`. Jednak w przypadku `std::vector<bool>` operator `operator[]` nie zwraca `bool&`. Zamiast tego zwraca całkiem nowy obiekt. Dlaczego i jak się to odbywa, zostało objaśnione w punkcie 6, ale teraz ważne jest to, że typ zwracany przez operator `operator[]` kontenera zależy od kontenera.

Instrukcja `decltype` sprawia, że możemy to łatwo wyrazić. Oto pierwsze podejście do szablonu, który chcemy napisać, pokazujące użycie instrukcji `decltype` do wyznaczenia zwracanego typu. Szablon wymaga odrobiny dopracowania, ale odłożymy to na później:

```
template<typename Container, typename Index>    // działa, ale
auto authAndAccess(Container& c, Index i)       // wymaga
  -> decltype(c[i])                             // dopracowania
{
  authenticateUser();
  return c[i];
}
```

Użycie auto przed nazwą funkcji nie ma nic wspólnego z dedukcją typów. Zamiast tego wskazuje, że użyta została składnia C++11 dla *opóźnionej deklaracji typu zwracanego*, co oznacza, że typ zwracany przez funkcję zostanie zadeklarowany za listą parametrów (za „->”). A opóźniona deklaracja typu zwracanego ma zaletę polegającą na tym, że parametry funkcji mogą być użyte do określenia zwracanego przez nią typu. Na przykład w funkcji authAndAccess określamy typ zwracany przy użyciu parametrów c i i. Jeżeli chcielibyśmy, aby typ zwracany poprzedzał nazwę funkcji w konwencjonalnym stylu, parametry c i i byłyby niedostępne, ponieważ nie byłyby jeszcze zadeklarowane.

Tak zadeklarowana funkcja authAndAccess zwraca dowolny typ zwracany przez operator operator[], gdy zostanie zastosowany do przekazanego kontenera, zgodnie z naszymi wymaganiami.

Standard C++11 dopuszcza dedukcję zwracanego typu dla wyrażeń lambda obejmujących jedną instrukcję, a standard C++14 rozszerza te możliwości na wszystkie wyrażenia lambda i wszystkie funkcje, w tym te z wieloma instrukcjami. W przypadku funkcji authAndAccess oznacza to, że w języku C++14 możemy pominąć opóźnioną deklarację typu zwracanego, pozostawiając tylko początkowe słowo kluczowe auto. W tej formie deklaracji auto *oznacza*, że ma miejsce dedukcja typów. W szczególności oznacza, że kompilatory wydedukują typ zwracany przez funkcję na podstawie jej implementacji:

```
template<typename Container, typename Index>    // C++14;
auto authAndAccess(Container& c, Index i)       // nie całkiem
{                                               // prawidłowo
  authenticateUser();
  return c[i];                  // zwracany typ dedukowany z c[i]
}
```

W punkcie 2 pokazałem, że dla funkcji ze zwracanym typem określonym jako auto kompilatory używają dedukcji typów w szablonie. W takim przypadku jest to problematyczne. Jak już mówiliśmy, operator operator[] dla większości kontenerów typu T zwraca T&, ale zgodnie z opisem w punkcie 1 wiemy, że podczas dedukcji typu w szablonie referencyjność inicjującego wyrażenia jest ignorowana. Rozważmy, co to oznacza dla tego kodu klienckiego:

```
std::deque<int> d;
…
authAndAccess(d, 5) = 10;  // uwierzytelnij użytkownika, zwróć d[5],
```

```
                                    // a następnie przypisz do tego 10;
                                    // to się nie skompiluje!
```

Tutaj `d[5]` zwraca `int&`, ale dedukcja typu zwracanego `auto` dla funkcji `authAndAccess` usunie referencyjność, w ten sposób sprawiając, że zwróconym typem będzie `int`. Ta wartość `int` zwracana przez funkcję jest r-wartością, więc powyższy kod próbuje przypisać 10 do r-wartości `int`. Jest to niedopuszczalne w języku C++, więc kod się nie skompiluje.

Aby funkcja `authAndAccess` działała tak, jak chcemy, musimy użyć dedukcji typu `decltype` dla jej typu zwracanego, aby określić, że funkcja `authAndAccess` powinna zwracać dokładnie ten sam typ, jaki zwraca wyrażenie `c[i]`. Obrońcy języka C++, przewidując potrzebę używania reguł dedukcji typów `decltype` w niektórych przypadkach, gdzie typy są wnioskowane, umożliwili to w standardzie C++14 za pomocą określenia `decltype(auto)`. To, co początkowo wydaje się sprzeczne (`decltype` *i* `auto`?), faktycznie ma doskonały sens: `auto` określa, że typ jest do wydedukowania, a `decltype` dopowiada, że podczas tej dedukcji mają być używane reguły `decltype`. Możemy dlatego napisać funkcję `authAndAccess` w taki sposób:

```
template<typename Container, typename Index>   // C++14; działa,
decltype(auto)                                 // ale nadal
authAndAccess(Container& c, Index i)           // wymaga dopracowania
{
  authenticateUser();
  return c[i];
}
```

Teraz funkcja `authAndAccess` będzie naprawdę zwracać to, co zwraca `c[i]`. W szczególności w typowym przypadku, gdy `c[i]` zwraca `T&`, funkcja `authAndAccess` również zwróci `T&`, a w nietypowym przypadku, gdy `c[i]` zwraca obiekt, funkcja `authAndAccess` również zwróci obiekt.

Użycie `decltype(auto)` nie jest ograniczone do typów zwracanych przez funkcje. Może być także wygodne do deklarowania zmiennych, gdy chcemy zastosować reguły dedukcji typów `decltype` do wyrażenia inicjującego:

```
Widget w;

const Widget& cw = w;
```