4.8. Wprowadzenie do szablonów i programowania uogólnionego

Widzieliśmy już klasy, które mogą działać z różnymi typami dostarczanymi podczas definiowania konkretnych obiektów. Najlepszym przykładem może być klasa `std::vector`, która może zostać zdefiniowana dla konkretnego typu obiektów do przechowywania, np. `std::vector< int >` do przechowywania `int`, `std::vector< std::string >` do przechowywania `std::string` itd. Jest to możliwe, ponieważ `std::vector` został zdefiniowany jako *klasa szablonowa* (ang. *template class*). W tym podrozdziale wyjaśnimy, co to oznacza i jak możemy definiować nasze własne klasy i funkcje szablonowe. Ściśle mówiąc, w tej części zapoznamy się z:

- Funkcjami i klasami szablonowymi.
- Sposobami konwertowania istniejącej funkcji lub klasy na bardziej ogólną wersję szablonową.
- Sposobami projektowania klasy szablonowej i organizacji kodu szablonu.
- Parametrami szablonu i sposobom ich dostarczania.
- Ograniczaniem dozwolonych parametrów szablonu z użyciem warunków.
- Specjalizacjami szablonów.
- Funkcjami składowymi szablonów i sposobie ich wykorzystywania.
- Dodatkowymi informacjami o `constexpr`, `static_assert` i wykonywaniu kodu w czasie kompilacji.

4.8.1. Uogólnianie klasy przy użyciu szablonów

W podrozdziale 4.4 omówiliśmy szczegóły klasy `TheCube`. Jest to samowystarczalna klasa, która jest w stanie przechować wartości typu `double`, zorganizowane w trójwymiarową kostkę danych. Z kolei w podrozdziale 4.4 zobaczyliśmy samowystarczalną klasę `TComplex`, która modeluje liczby zespolone. Implementuje ona również kilka operacji specyficznych dla dziedziny liczb zespolonych. Wyobraźmy sobie teraz, że chcemy utworzyć trójwymiarową kostkę danych, ale zamiast `double` chcemy przechowywać w nim wartości typu `TComplex`. Co możemy zrobić? Najprostszym rozwiązaniem i jednocześnie jedynym dostępnym w językach takich jak C jest skopiowanie kodu `TheCube`, wklejenie go i nadanie mu innej nazwy (przykładowo `TComplexCube`), a następnie zamiana wszystkich wystąpień `double` na `TComplex`. Później, jeśli będzie nam potrzebna kostka tekstu, możemy powtórzyć tę procedurę, zamieniając tym razem wystąpienia `double` na `string` itd. Niestety, jeśli będziemy postępować w ten sposób, otrzymamy wiele niemal identycznych klas, różniących się tylko w kilku miejscach, w których pojawia się typ `double`. Co, jeśli zdecydujemy się wtedy usprawnić niewielki fragment kodu? Kuszące wydaje się utworzenie wspólnego wzorca: szablonu, w którym będziemy mogli wskazać miejsca do zmiany. Gdy obiekt takiej klasy będzie instancjonowany, będziemy mogli podać konkretny typ do wykorzystania w tych miejscach. Taki mechanizm istnieje w C++ i nazywany jest *klasami szablonowymi*. Pozwala nam on definiować klasy, które są wystarczająco uniwersalne, aby mogły działać z obiektami dowolnego typu – nawet takiego, który

dopiero utworzymy. Takie podejście programistyczne, które pasuje (prawie) do każdego typu, nazywamy *programowaniem uogólnionym* (ang. *generic programming*). W podobny sposób możemy utworzyć funkcje szablonowe – jako funkcje samodzielne lub składowe klasy (punkt 4.8.5).

W tym punkcie nauczymy się:

- Konwertowania nieszablonowej klasy do jej szablonowej wersji.
- Tworzenia instancji szablonu.

Aby zobaczyć, w jaki sposób możemy utworzyć klasę ogólną z użyciem mechanizmu szablonów C++, napiszmy szablonową wersję klasy `TheCube`, której definicja widnieje w tabeli 4.3. Aby ułatwić to zadanie, popracujmy z mniejszą wersją tej klasy o nazwie `TinyCube`, zdefiniowanej w listingu 4.12.

Listing 4.12. Definicja `TinyCube` z podkreślonymi specyficznymi typami (w *CppBookCode*, *TinyCube.cpp*)

```
class TinyCube
{
   public:

      static const int kDims = 3;  // to samo dla wszystkich obiektów tej klasy

      enum EDims { kx, ky, kz };   // skróty dla 3 wymiarów

   private:

      vector< double >     fDataBuf;  // wektor do przechowywania danych

      array< int, kDims >  fDim;     // przechowuje zakres każdego wymiaru

   public:

      // Konstruktor parametryczny - dx, dy, dz muszą być > 0
      TinyCube( const int dx, const int dy, const int dz )
         : fDim{ dx, dy, dz }, fDataBuf( dx * dy * dz, 0.0 )
      {
         assert( dx > 0 && dy > 0 && dz > 0 );
         assert( fDataBuf.size() == dx * dy * dz );
      }

      // Destruktor niczego nie robi. Dane zostaną usunięte przez wektor, jednak
      // ~TinyCube() {} // jawna definicja destruktora wykluczałaby semantykę przekazywania

   public:

      // Uzyskaj dostęp do elementów przez referencję - dwukierunkowe
      auto & Element( const int x, const int y, const int z )
      {
         const auto offset = ( z * fDim[ ky ] + y ) * fDim[ kx ] + x;
         return fDataBuf[ offset ];  // indeks zwracany przez referencję
      }

};
```