Siddhartha Rao

Programowanie w Twoim zasięgu!

Wydanie VII

C++

Dla **każdego**

SAMS

Helion

Tytuł oryginału: Sams Teach Yourself C++ in One Hour a Day (7th Edition)

Tłumaczenie: Robert Górczyński

ISBN: 978-83-246-8077-1

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/cppit7.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cppit7
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Lekcja 8

Wskaźniki i referencje

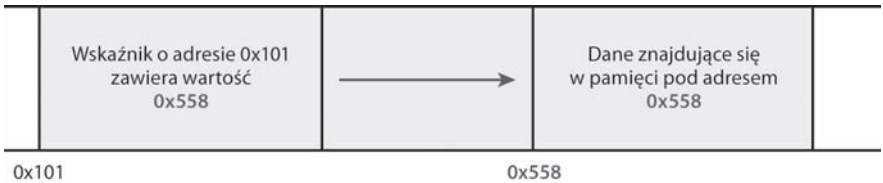
Jedną z największych zalet języka C++ jest możliwość tworzenia aplikacji wysokiego poziomu, które będą działały niezależnie od komputera. Język C++ pozwala na dostrojenie wydajności aplikacji na poziomie bitów i bajtów. Zrozumienie sposobu działania wskaźników i referencji to ważny krok na drodze do zdobycia umiejętności tworzenia programów efektywnie wykorzystujących dostępne zasoby systemu.

Z tej lekcji dowiesz się:

- **czym są wskaźniki,**
- **czym jest pula wolnej pamięci,**
- **jak używać operatorów new i delete w celu alokacji i zwalniania pamięci,**
- **jak tworzyć stabilne aplikacje za pomocą wskaźników i alokacji dynamicznej,**
- **czym są referencje,**
- **na czym polegają różnice pomiędzy wskaźnikami i referencjami,**
- **kiedy używać wskaźników, a kiedy referencji.**

Czym jest wskaźnik?

Ujmując rzecz najprościej, *wskaźnik* to zmienna przechowująca adres w pamięci. Podobnie jak zmienna typu `int` jest przeznaczona do przechowywania wartości w postaci liczby całkowitej, wskaźnik przechowuje adres w pamięci, co pokazano na rysunku 8.1.



RYSUNEK 8.1. Wizualne przedstawienie wskaźnika

Wskaźnik jest więc zmienną i podobnie jak wszystkie zmienne zabiera pewną ilość pamięci (w przypadku pokazanym na rysunku 8.1 będzie to pamięć pod adresem 0x101). Cechą charakterystyczną wskaźnika jest fakt, że przechowywana w nim wartość (tutaj 0x558) jest interpretowana jako adres w pamięci. Dlatego też wskaźnik jest zmienną specjalną *wskazującą* położenie w pamięci.

Deklaracja wskaźnika

Wskaźnik jest zmienną, więc musi być zadeklarowany. Zwykle deklarujesz wskaźnik, aby prowadzić do wartości określonego typu (np. `int`). Oznacza to, że adres przechowywany we wskaźniku prowadzi do miejsca w pamięci, w którym znajduje się liczba całkowita. Istnieje również możliwość określenia wskaźnika w taki sposób, aby wskazywał blok pamięci (to wskaźnik `void`).

Wskaźnik będący zmienną musi być zadeklarowany w sposób podobny do sposobu deklarowania pozostałych zmiennych:

```
TypWskaźnika * NazwaZmiennejWskaźnika;
```

Tak jak w większości zmiennych, jeśli nie zainicjalizujesz wskaźnika, będzie zawierał losowo wybraną wartość. Ponieważ nie chcesz, aby program uzyskał dostęp do losowo wybranego położenia w pamięci, musisz zainicjalizować wskaźnik z wartością `null`. To `null` jest wartością, którą można wykorzystać w operacji sprawdzania i która jednocześnie nie jest adresem w pamięci:

```
TypWskaźnika * NazwaZmiennejWskaźnika = NULL; // Inicjalizacja wartości wskaźnika.
```

Deklaracja wskaźnika typu liczby całkowitej będzie więc miała postać:

```
int *pInteger = NULL;
```

Ostrzeżenie

Wskaźnik, podobnie jak wszystkie poznane dotąd typy danych, zawiera przypadkową wartość dopóty, dopóki nie zostanie zainicjalizowany. Taka przypadkowa wartość jest szczególnie niebezpieczna w przypadku wskaźników, ponieważ wskazuje pewien adres w pamięci. Niezainicjalizowany wskaźnik może spowodować, że program uzyska dostęp do niepoprawnego położenia w pamięci i ulegnie awarii.

Określenie adresu zmiennej przy użyciu operatora referencji (&)

Zmienne to narzędzia dostarczane przez język, aby umożliwić pracę z danymi w pamięci. Koncepcja ta została szczegółowo wyjaśniona w lekcji 3., zatytułowanej „Zmienne i stałe". Wskaźniki również są zmiennymi, ale mają specjalny typ używany jedynie w celu przechowywania adresu w pamięci.

Jeżeli `VarName` to nazwa zmiennej, wtedy `&VarName` podaje adres w pamięci, gdzie przechowywana jest wartość wymienionej zmiennej.

Jeśli zatem zadeklarowałeś liczbę całkowitą, używając przedstawionej poniżej doskonale znanej składni:

```
int Age = 30;
```

wówczas &Age będzie adresem w pamięci przechowującym wartość (30) zmiennej. W listingu 8.1 przedstawiono koncepcję adresu w pamięci dla zmiennej w postaci liczby całkowitej, który jest używany do przechowywania wartości zmiennej.

Listing 8.1. Określenie adresu liczb typu int i double

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    int Age = 30;
6:    const double Pi = 3.1416;
7:
8:    // Użycie operatora & do wyszukania adresu w pamięci.
9:    cout << "Liczba całkowita Age jest pod adresem: 0x" << hex << &Age <<
      ↪endl;
```

```
10:    cout << "Liczba Pi typu double jest pod adresem: 0x" << hex << &Pi <<
       ↪endl;
11:
12:    return 0;
13: }
```

Wynik ▼

```
Liczba całkowita Age jest pod adresem: 0x0045FE00
Liczba Pi typu double jest pod adresem: 0x0045FDF8
```

Analiza ▼

Zwróć uwagę na użycie operatora referencji (&) w wierszach 9. i 10. w celu ustalenia adresów zmiennej `Age` i stałej `Pi`. Przedrostek `0x` został dodany, aby zachować konwencję stosowaną podczas wyświetlania liczb szesnastkowych.

Uwaga

Wiesz już, że ilość pamięci zużywana przez zmienną zależy od jej typu. W listingu 3.4 przedstawionym w lekcji 3. użyto operatora `sizeof()` w celu pokazania, że wielkość liczby całkowitej to 4 bajty (w systemie autora i używanym przez niego kompilatorze). Opierając się na przedstawionych powyżej danych wyjściowych wskazujących na położenie zmiennej `Age` pod adresem `0x0045FE08` i znając wielkość liczby całkowitej (4 bajty), wiemy, że cztery bajty w położeniu od `0x0045FE00` do `0x0045FE04` należą do liczby całkowitej `Age`.

Uwaga

Operator referencji (&) jest nazywany również operatorem adresu.

Użycie wskaźników do przechowywania adresów

Dowiedziałeś się już, jak deklarować wskaźniki oraz jak ustalić adres zmiennej. Wiesz także, że wskaźniki są zmiennymi używanymi do przechowywania adresów w pamięci. Najwyższy czas skonsolidować wiedzę i wykorzystać wskaźniki do przechowywania adresów pobranych przy użyciu operatora referencji (&).

Przyjmujemy założenie, że deklaracja zmiennej jest przeprowadzana w znany sposób:

```
// Deklaracja zmiennej.
Typ NazwaZmiennej = WartośćPoczątkowa;
```

W celu przechowywania adresu zmiennej we wskaźniku konieczne jest zadeklarowanie wskaźnika o takim samym typie jak zmienna (Typ) oraz zainicjalizowanie wskaźnika przy użyciu operatora referencji (&) wraz z adresem w pamięci, gdzie znajduje się wartość zmiennej:

```
// Zadeklarowanie wskaźnika takiego samego typu i zainicjalizowanie go z adresem zmiennej
// w pamięci.
Typ* Wskaźnik = &Zmienna;
```

Dlatego też w przypadku zadeklarowania zmiennej w postaci liczby całkowitej i przy użyciu doskonale znanej składni:

```
int Age = 30;
```

możesz zadeklarować wskaźnik typu int zawierający rzeczywisty adres w pamięci, gdzie przechowywana jest wartość zmiennej Age:

```
int* pInteger = &Age; // Wskaźnik do liczby całkowitej Age.
```

W listingu 8.2 możesz zobaczyć, jak wskaźnik jest używany do przechowywania adresu uzyskanego za pomocą operatora referencji (&).

Listing 8.2. Przykład deklaracji i inicjalizacji wskaźnika

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Age = 30;
 6:    int* pInteger = &Age;   // Wskaźnik typu int, zainicjalizowany z wartością &Age.
 7:
 8:    // Wyświetlenie wartości wskaźnika.
 9:    cout << "Liczba całkowita Age jest pod adresem: 0x" << hex <<
       ↪pInteger << endl;
10:
11:    return 0;
12: }
```

Wynik ▼

```
Liczba całkowita Age jest pod adresem: 0x0045FE00
```

Analiza ▼

Dane wyjściowe powyższego listingu są w zasadzie takie same jak poprzedniego, w obu przypadkach został wyświetlony ten sam komunikat wraz z adresem

w pamięci, gdzie przechowywana jest wartość zmiennej Age. Różnica w stosunku do poprzedniego programu polega na tym, że adres jest najpierw przypisywany wskaźnikowi (wiersz 6.), a następnie wartość wskaźnika (teraz już adres) zostaje wyświetlona przy użyciu polecenia cout (wiersz 9.).

Uwaga

> Dane wyjściowe, które otrzymasz, mogą się różnić od przedstawionych w książce. Adres zmiennej może być inny po każdym uruchomieniu programu w tym samym komputerze.

Skoro już wiesz, jak przechowywać adres w zmiennej wskaźnika, bardzo łatwo możesz wyobrazić sobie, że tej samej zmiennej wskaźnika można przypisać inny adres w pamięci. W ten sposób wskaźnik będzie prowadził do innej wartości, co przedstawiono w listingu 8.3.

Listing 8.3. Przypisanie wskaźnikowi adresu innej zmiennej

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Age = 30;
 6:
 7:    int* pInteger = &Age;
 8:    cout << "pInteger prowadzi teraz do zmiennej Age" << endl;
 9:
10:    // Wyświetlenie wartości wskaźnika.
11:    cout << "pInteger = 0x" << hex << pInteger << endl;
12:
13:    int DogsAge = 9;
14:    pInteger = &DogsAge;
15:    cout << "pInteger prowadzi teraz do zmiennej DogsAge" << endl;
16:
17:    cout << "pInteger = 0x" << hex << pInteger << endl;
18:
19:    return 0;
20: }
```

Wynik ▼

```
pInteger prowadzi teraz do zmiennej Age
pInteger = 0x002EFB34
pInteger prowadzi teraz do zmiennej DogsAge
pInteger = 0x002EFB1C
```

Analiza ▼

Program pokazuje, że wskaźnik (pInteger) prowadzący do jednej liczby całkowitej może prowadzić także do zupełnie innej liczby całkowitej. W wierszu 7. nastąpiła jego inicjalizacja wraz z wartością &Age, a więc zawierał adres zmiennej Age. W wierszu 14. temu samemu wskaźnikowi przypisano wartość &DogsAge, co oznacza, że prowadzi do innego położenia w pamięci, które zawiera wartość zmiennej DogsAge. Dane wyjściowe pokazują, że wartość wskaźnika (tzn. adres w pamięci) uległa zmianie. Najpierw prowadziła do miejsca w pamięci (0x002EFB34) przechowującego liczbę całkowitą Age, a później do innego miejsca (0x002EFB1C) przechowującego liczbę całkowitą DogsAge.

Uzyskanie dostępu do danych przy użyciu operatora dereferencji (*)

Masz wskaźnik do danych zawierający poprawny adres. W jaki sposób można uzyskać dostęp do wspomnianego położenia, tzn. jak pobrać lub zmienić dane we wskazanym położeniu w pamięci? Odpowiedzią jest użycie operatora dereferencji (*) nazywanego także operatorem wyłuskania. Jeżeli masz prawidłowy wskaźnik pData, wtedy użycie *pData pozwala na uzyskanie dostępu do wartości przechowywanej pod adresem wskazywanym przez wskaźnik. Praktyczne zastosowanie operatora dereferencji przedstawiono w listingu 8.4.

Listing 8.4. Przykład użycia operatora dereferencji (*) w celu uzyskania dostępu do wartości wskazywanej przez wskaźnik

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Age = 30;
 6:    int DogsAge = 9;
 7:
 8:    cout << "Liczba całkowita Age = " << Age << endl;
 9:    cout << "Liczba całkowita DogsAge = " << DogsAge << endl;
10:
11:    int* pInteger = &Age;
12:    cout << "pInteger prowadzi do zmiennej Age" << endl;
13:
14:    // Wyświetlenie wartości wskaźnika.
```

```
15:    cout << "pInteger = 0x" << hex << pInteger << endl;
16:
17:    // Wyświetlenie wartości we wskazanym położeniu.
18:    cout << "*pInteger = " << dec << *pInteger << endl;
19:
20:    pInteger = &DogsAge;
21:    cout << "pInteger prowadzi teraz do zmiennej DogsAge" << endl;
22:
23:    cout << "pInteger = 0x" << hex << pInteger << endl;
24:    cout << "*pInteger = " << dec << *pInteger << endl;
25:
26:    return 0;
27: }
```

Wynik ▼

```
Liczba całkowita Age = 30
Liczba całkowita DogsAge = 9
pInteger prowadzi do zmiennej Age
pInteger = 0x0025F788
*pInteger = 30
pInteger prowadzi teraz do zmiennej DogsAge
pInteger = 0x0025F77C
*pInteger = 9
```

Analiza ▼

Poza zmianą adresu przechowywanego przez wskaźnik, co również miało miejsce w poprzednim przykładzie (listing 8.3), w powyższym listingu użyto operatora dereferencji (*) wraz ze zmienną wskaźnika `pInteger` w celu wyświetlenia różnych wartości przechowywanych w dwóch adresach w pamięci. Zwróć uwagę na wiersze 18. i 24.

W obu tych wierszach dostęp do liczby całkowitej wskazywanej przez wskaźnik `pInteger` następuje za pomocą operatora dereferencji. Po zmianie adresu we wskaźniku `pInteger` (patrz wiersz 20.) ten sam wskaźnik prowadzi teraz do wartości zmiennej `DogsAge`, co powoduje wyświetlenie w danych wyjściowych liczby 9.

Podczas użycia operatora dereferencji adres przechowywany we wskaźniku jest wykorzystywany przez aplikację do pobrania czterech bajtów z pamięci zarezerwowanej dla danej liczby całkowitej (ponieważ to wskaźnik do liczby całkowitej o wielkości 4 bajtów, co potwierdza wynik operacji `sizeof(int)`). Dlatego też zapewnienie poprawności adresu przechowywanego przez wskaźnik

ma znaczenie krytyczne. Przez inicjalizację wskaźnika z wartością &Age (wiersz 11.) gwarantujemy, że wskaźnik zawiera prawidłowy adres w pamięci. Jeżeli nie zainicjalizujesz wskaźnika, może on zawierać dowolną wartość, która istniała w danym położeniu w pamięci w chwili rezerwacji tej pamięci dla wskaźnika. Odwołanie do takiego wskaźnika najczęściej oznacza wystąpienie błędu *Access Violation*, czyli próbę uzyskania przez aplikację dostępu do adresu w pamięci, do którego nie ma uprawnień.

Operator dereferencji jest nazywany również operatorem dostępu pośredniego.

Uwaga

W poprzednim przykładzie wskaźnik był używany w celu odczytu (pobrania) wartości ze wskazanego miejsca w pamięci. W listingu 8.5 pokazano, co się stanie, jeśli wskaźnik `*pInteger` zostanie użyty jako l-wartość, tzn. kod przypisze go zmiennej, zamiast uzyskać do niego dostęp.

Listing 8.5. Operowanie danymi za pomocą wskaźnika i operatora dereferencji

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int DogsAge = 30;
 6:    cout << "Zainicjalizowana zmienna DogsAge = " << DogsAge << endl;
 7:
 8:    int* pAge = &DogsAge;
 9:    cout << "Wskaźnik pAge prowadzi do zmiennej DogsAge" << endl;
10:
11:    cout << "Podaj wiek psa: ";
12:
13:    // Dane wejściowe zostają umieszczone w pamięci wskazywanej przez wskaźnik pAge.
14:    cin >> *pAge;
15:
16:    // Wyświetlenie adresu, gdzie przechowywana jest wartość.
17:    cout << "Dane wejściowe wskazywane przez pAge są pod adresem 0x" <<
       ↪hex << pAge << endl;
18:
19:    cout << "Liczba całkowita DogsAge = " << dec << DogsAge << endl;
20:
21:    return 0;
22: }
```

Wynik ▼

```
Zainicjalizowana zmienna DogsAge = 30
Wskaźnik pAge prowadzi do zmiennej DogsAge
Podaj wiek psa: 10
Dane wejściowe wskazywane przez pAge są pod adresem 0x0025FA18
Liczba całkowita DogsAge = 10
```

Analiza ▼

W omawianym listingu kluczowy krok to wiersz 14., w którym liczba całkowita podana przez użytkownika zostaje zapisana w położeniu w pamięci wskazywanym przez wskaźnik `pAge`. Zauważ, że pomimo umieszczenia danych wejściowych w położeniu wskazywanym przez `pAge`, polecenie w wierszu 19. wyświetlające wartość zmiennej `DogsAge` pokazuje wartość wskazaną przez wskaźnik. Wynika to z faktu, że wskaźnik `pAge` prowadzi do zmiennej `DogsAge`, na co wskazuje polecenie inicjalizujące w wierszu 8. Wszelkie zmiany w położeniach w pamięci przechowujących zmienną `DogsAge` i wskazywanych przez wskaźnik `pAge` są odzwierciedlane w obu elementach.

Ile pamięci zabiera wskaźnik?

Dowiedziałeś się, że wskaźnik to po prostu rodzaj zmiennej przechowującej adres położenia w pamięci. Dlatego też, niezależnie od wskazywanego typu, zawartością wskaźnika jest adres, czyli liczba. Wielkość adresu to liczba bajtów wymaganych do przechowywania liczby w postaci stałej w danym systemie. Wynik działania operatora `sizeof()` względem wskaźnika będzie zależał od kompilatora i systemu operacyjnego, dla którego program został skompilowany. Natomiast nie zależy od natury danych, do których prowadzi wskaźnik, co potwierdza program przedstawiony w listingu 8.6.

Listing 8.6. Potwierdzenie, że wskaźniki prowadzące do różnych typów danych mają tę samą wielkość

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    int Age = 30;
6:    double Pi = 3.1416;
7:    char SayYes = 'y';
```

```
 8:
 9:    // Inicjalizacja wskaźników wraz z adresami zmiennych.
10:    int* pInt = &Age;
11:    double* pDouble = &Pi;
12:    char* pChar = &SayYes;
13:
14:    cout << "sizeof podstawowych typów -" << endl;
15:    cout << "sizeof(int) = " << sizeof(int) << endl;
16:    cout << "sizeof(double) = " << sizeof(double) << endl;
17:    cout << "sizeof(char) = " << sizeof(char) << endl;
18:
19:    cout << "sizeof wskaźników do podstawowych typów -" << endl;
20:    cout << "sizeof(pInt) = " << sizeof(pInt) << endl;
21:    cout << "sizeof(pDouble) = " << sizeof(pDouble) << endl;
22:    cout << "sizeof(pChar) = " << sizeof(pChar) << endl;
23:
24:    return 0;
25: }
```

Wynik ▼

```
sizeof podstawowych typów -
sizeof(int) = 4
sizeof(double) = 8
sizeof(char) = 1
sizeof wskaźników do podstawowych typów -
sizeof(pInt) = 4
sizeof(pDouble) = 4
sizeof(pChar) = 4
```

Analiza ▼

Dane wyjściowe jasno pokazują, że nawet gdy wartość `sizeof(char)` wynosi 1 bajt, a `sizeof(double)` wynosi 8 bajtów, to wielkość wskaźnika (`sizeof(pointer)`) jest stała i wynosi 4 bajty. Po prostu ilość pamięci wymagana do przechowywania adresów w pamięci jest taka sama, niezależnie od tego, czy wskaźnik prowadzi do 1, czy do 8 bajtów.

Uwaga

Dane wyjściowe listingu 8.6 pokazują, że wielkość wskaźnika wynosi 4 bajty. Jednak w Twoim systemie operacyjnym wielkość ta może być inna. Przedstawione w książce dane wyjściowe zostały wygenerowane przez kod skompilowany przez 32-bitowy kompilator. Jeżeli używasz 64-bitowego kompilatora i uruchomisz program w 64-bitowym systemie operacyjnym, możesz zobaczyć, że wielkość zmiennej wskaźnika wynosi 64 bity, czyli 8 bajtów.

Dynamiczna alokacja pamięci

Kiedy tworzysz program zawierający deklarację tablicy poniższego typu:

```
int Numbers[100]; // Tablica statyczna stu liczb całkowitych.
```

program boryka się z dwoma problemami. Oto one.

1. Ograniczasz pojemność programu, który będzie mógł przechowywać maksymalnie sto liczb całkowitych.
2. Zmniejszasz wydajność systemu, w przypadku gdy tablica zarezerwowana dla stu liczb całkowitych będzie przechowywała tylko jedną.

Wymienione problemy istnieją z powodu alokacji pamięci dla tablicy — dla podanej wcześniej deklaracji tablicy kompilator rezerwuje statyczną ilość pamięci.

Aby utworzyć aplikację optymalnie wykorzystującą zasoby pamięci na podstawie potrzeb i działań użytkownika, należy zastosować dynamiczną alokację pamięci. W ten sposób będzie można zaalokować większą ilość pamięci, gdy będzie potrzebna, oraz zwalniać pamięć, kiedy nie będziemy z niej korzystać. Język C++ oferuje dwa operatory — `new` i `delete` — pomagające w lepszym zarządzaniu pamięcią przez aplikację. Wskaźniki będące zmiennymi używanymi do przechowywania adresów w pamięci odgrywają krytyczną rolę w efektywnym, dynamicznym zarządzaniu pamięcią.

Użycie operatorów new i delete w celu dynamicznej alokacji i zwalniania pamięci

Operator `new` służy do alokacji nowych bloków pamięci. Najczęściej używana forma operatora `new` powoduje zwrot wskaźnika do żądanej pamięci, o ile operacja zakończyła się powodzeniem, w przeciwnym razie następuje zgłoszenie wyjątku. Podczas użycia operatora `new` konieczne jest wskazanie typu danych, dla których następuje alokacja pamięci:

```
Typ* Wskaźnik = new Typ; // Żądanie pamięci dla jednego elementu.
```

Istnieje również możliwość wskazania liczby elementów, dla których ma być zaalokowana pamięć (w przypadku alokacji pamięci dla więcej niż tylko jednego elementu):

```
Typ* Wskaźnik = new Typ[LiczbaElementów]; // Żądanie pamięci dla podanej liczby
                                          // elementów.
```

Dlatego też, jeśli musisz zaalokować pamięć dla liczb całkowitych, możesz skorzystać z poniższych składni:

```
int* pNumber = new int;  // Pobranie wskaźnika do liczby całkowitej.
int* pNumbers = new int[10];  // Pobranie wskaźnika do bloku dziesięciu liczb całkowitych.
```

> Zauważ, że new oznacza żądanie pamięci. Nie ma gwarancji, że wykonanie polecenia żądania pamięci zakończy się powodzeniem — zależy to od stanu systemu oraz dostępności zasobów pamięci.
>
> **Uwaga**

Dla każdej przeprowadzonej operacji alokacji pamięci za pomocą operatora new konieczne jest przeprowadzenie operacji zwolnienia tej pamięci przy użyciu operatora delete:

```
Typ* Wskaźnik = new Typ;
delete Wskaźnik;  // Zwolnienie pamięci zaalokowanej wcześniej dla jednego egzemplarza wskazanego typu.
```

Ta sama reguła obowiązuje w przypadku żądania alokacji pamięci dla wielu elementów:

```
Typ* Wskaźnik = new Typ[LiczbaElementów];
delete[] Wskaźnik; // Zwolnienie zaalokowanego powyżej bloku pamięci.
```

> Zwróć uwagę na użycie delete[] w czasie alokacji bloku pamięci przy użyciu operatora new[...] i delete po alokacji tylko jednego elementu za pomocą operatora new.
>
> **Uwaga**

Jeżeli nie zwolnisz zaalokowanej wcześniej pamięci, której już nie używasz, pozostanie zarezerwowana dla aplikacji. To z kolei zmniejszy ilość pamięci systemowej dostępnej dla innych programów i prawdopodobnie zmniejszy szybkość działania Twojej aplikacji. Taka sytuacja nosi nazwę *wycieku pamięci*; należy jej unikać za wszelką cenę.

Kod przedstawiony w listingu 8.7 pokazuje dynamiczną alokację i zwalnianie pamięci.

Listing 8.7. Wykorzystanie operatora dereferencji w celu uzyskania dostępu do pamięci zaalokowanej przez operator new oraz jej zwolnienie za pomocą operatora delete

```
0: #include <iostream>
1: using namespace std;
2:
```

```
 3: int main()
 4: {
 5:     // Żądanie alokacji pamięci dla elementu typu int.
 6:     int* pAge = new int;
 7:
 8:     // Użycie zaalokowanej pamięci do przechowywania liczby.
 9:     cout << "Podaj wiek psa: ";
10:     cin >> *pAge;
11:
12:     // Użycie operatora dereferencji * w celu uzyskania dostępu do przechowywanej wartości.
13:     cout << "Wartość zmiennej Age wynosi " << *pAge << " i jest
        ↪przechowywana pod adresem 0x" << hex << pAge << endl;
14:
15:     delete pAge; // Zwolnienie pamięci.
16:
17:     return 0;
18: }
```

Wynik ▼

```
Podaj wiek psa: 9
Wartość zmiennej Age wynosi 9 i jest przechowywana pod adresem 0x00338120
```

Analiza ▼

W wierszu 6. pokazano przykład użycia operatora `new` w celu alokacji pamięci dla liczby całkowitej. W zarezerwowanej pamięci będą przechowywane podane przez użytkownika dane wejściowe, czyli wiek psa. Zauważ, że operator `new` zwraca wskaźnik i to jest powodem przypisania mu wartości. Wiek podany przez użytkownika jest przechowywany w nowo zaalokowanej pamięci, odbywa się to w wierszu 10. za pomocą polecenia `cin` i operatora dereferencji (*). W wierszu 13. kod wyświetla przechowywaną wartość (ponownie posługując się operatorem dereferencji), a także adres w pamięci, pod którym znajduje się przechowywana wartość. Zwróć uwagę, że adres znajdujący się we wskaźniku `pAge` (wiersz 13.) to ten sam adres, który został zwrócony przez operator `new` w wierszu 6. i od tamtej chwili nie uległ zmianie.

Ostrzeżenie

Operator `delete` nie może być wywoływany względem dowolnego adresu znajdującego się we wskaźniku, to musi być adres zwrócony wcześniej przez operator `new` i tylko ten, który nie był wcześniej zwolniony przez inne wywołanie `delete`.

Mimo iż wskaźniki przedstawione w listingu 8.6 zawierają poprawne adresy, jednak nie powinny być zwalniane za pomocą operatora `delete`, ponieważ wspomniane adresy nie zostały otrzymane na skutek wywołania operatora `new`.

Pamiętaj, że po użyciu `new[...]` w celu alokacji bloku pamięci dla wielu elementów zwolnienie pamięci musi nastąpić przez wywołanie `delete[]`, jak to przedstawiono w listingu 8.8.

Listing 8.8. Alokacja pamięci przy użyciu operatora new[...] i jej zwolnienie za pomocą operatora delete[]

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: int main()
 5: {
 6:    cout << "Podaj imię: ";
 7:    string Name;
 8:    cin >> Name;
 9:
10:    // Dodanie 1 do rezerwowanej pamięci (na znak null).
11:    int CharsToAllocate = Name.length() + 1;
12:
13:    // Żądanie pamięci potrzebnej do przechowywania kopii danych wejściowych.
14:    char* CopyOfName = new char [CharsToAllocate];
15:
16:    // Funkcja strcpy() kopiuje dane z ciągu tekstowego zakończonego znakiem null.
17:    strcpy(CopyOfName, Name.c_str());
18:
19:    // Wyświetlenie skopiowanego ciągu tekstowego.
20:    cout << "Dynamicznie zaalokowany bufor zawiera: " << CopyOfName <<
       ↪endl;
21:
22:    // Po zakończeniu pracy z buforem należy go usunąć.
23:    delete[] CopyOfName;
24:
25:    return 0;
26: }
```

Wynik ▼

```
Podaj imię: Robert
Dynamicznie zaalokowany bufor zawiera: Robert
```

Analiza ▼

Najbardziej interesujące nas wiersze w powyższym programie to te, które zawierają operatory `new` i `delete[]`, czyli (odpowiednio) wiersze 11. i 23. Różnica w stosunku do poprzedniego przykładu (patrz listing 8.7) polega na alokacji bloku pamięci wystarczającego do przechowywania wielu elementów — kod w listingu 8.7 alokował pamięć tylko dla jednego elementu. Alokacji pamięci dla tablicy elementów musi odpowiadać użycie operatora `delete[]` w celu zwolnienia pamięci po zakończeniu pracy z tablicą. Liczba znaków, dla których trzeba zaalokować pamięć, jest obliczana w wierszu 11. Liczba ta jest o jeden większa od liczby znaków wprowadzonych przez użytkownika, ponieważ trzeba uwzględnić znak `null`, bardzo ważny w ciągach tekstowych typu C. Konieczność umieszczenia znaku `null` na końcu ciągu tekstowego została wyjaśniona w lekcji 4., zatytułowanej „Tablice i ciągi tekstowe". Rzeczywista operacja kopiowania odbywa się w wierszu 17., jest przeprowadzana przez funkcję `strcpy()` używającą z kolei funkcji `c_str()` dostarczanej przez klasę `std::string`. Znaki są kopiowane do bufora typu `char` o nazwie `CopyOfName`.

Uwaga
Uwaga

> Operatory `new` i `delete` alokują pamięć pochodzącą z puli wolnej pamięci. Pula wolnej pamięci to rodzaj abstrakcji w postaci puli pamięci, którą aplikacja może zaalokować (tzn. zarezerwować) oraz dealokować (tzn. zwalniać).

Efektywne użycie operatorów inkrementacji (++) i dekrementacji (--) na wskaźnikach

Wskaźnik zawiera adres w pamięci. Przykładowo w przedstawionym wcześniej listingu 8.3 wskaźnik prowadzący do liczby całkowitej zawiera wartość `0x002EFB34`, czyli adres w pamięci, gdzie jest przechowywana wspomniana liczba całkowita. Sama liczba ma wielkość czterech bajtów, a tym samym zabiera cztery miejsca w pamięci od `0x002EFB34` do `0x002EFB37`. Inkrementacja wskaźnika za pomocą operatora ++ *nie spowoduje*, że wskaźnik będzie prowadził do `0x002EFB35`. Takie działanie, które ma na celu wskazanie środka liczby całkowitej, jest dosłownie bezcelowe.

Operacja inkrementacji lub dekrementacji wskaźnika jest interpretowana przez kompilator jako potrzeba wskazania kolejnej wartości w bloku pamięci, przy przyjęciu założenia, że będzie ona takiego samego typu, a *nie następnym bajtem* (o ile wartość nie ma długości jednego bajta, np. `char`).

Dlatego też inkrementacja wskaźnika, takiego jak `pInteger`, użytego w listingu 8.3 powoduje inkrementację o cztery bajty, co odpowiada wynikowi wywołania `sizeof(int)`. Użycie operatora ++ względem wskaźnika oznacza poinformowanie kompilatora, że wskaźnik ma prowadzić do kolejnej liczby całkowitej. Po operacji inkrementacji wskaźnik będzie prowadził do adresu 0x002EFB38. Podobnie dodanie wartości 2 do wskaźnika spowoduje jego przesunięcie o dwie liczby całkowite do przodu, czyli o osiem bajtów. Dalej w tej lekcji zobaczysz korelację pomiędzy tym zachowaniem wskaźników a indeksami używanymi w tablicach.

Dekrementacja wskaźników przy użyciu operatora `--` daje taki sam efekt, tzn. wartość adresu zawartego we wskaźniku zostaje zmniejszona o wynik operacji `sizeof` przeprowadzonej na typie danych wskazywanych przez wskaźnik.

Co się stanie po inkrementacji lub dekrementacji wskaźnika?
Adres znajdujący się we wskaźniku jest inkrementowany lub dekrementowany o wartość zwrotną działania operatora `sizeof` względem typu danych wskazywanego przez wskaźnik (to niekoniecznie będzie bajt). Dzięki temu kompilator gwarantuje, że wskaźnik nigdy nie będzie prowadzić do środka lub na koniec danych umieszczonych pod danym adresem w pamięci — wskaźnik zawsze prowadzi na początek danych.

Jeżeli wskaźnik zostałby zadeklarowany w postaci:

```
Typ* pTyp = Adres;
```

wówczas `++pTyp` oznacza, że `pTyp` zawiera (a więc wskazuje) wartość `Adres + sizeof(Typ)`.

W kodzie przedstawionym w listingu 8.9 znajdziesz wyjaśnienie efektu inkrementacji wskaźnika i dodawania do niego wartości przesunięcia.

Listing 8.9. Alokacja dynamiczna w zależności od potrzeb, inkrementacja wskaźników za pomocą wartości przesunięcia i operatora ++

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:   cout << "Ile liczb całkowitych chcesz podać? ";
6:   int InputNums = 0;
7:   cin >> InputNums;
8:
9:   int* pNumbers = new int [InputNums]; // Alokacja żądanych liczb całkowitych.
```

```
10:   int* pCopy = pNumbers;
11:
12:   cout<<"Udało się zaalokować pamięć dla "<<InputNums<< " liczb
      ↪całkowitych"<<endl;
13:   for(int Index = 0; Index < InputNums; ++Index)
14:   {
15:      cout << "Podaj liczbę "<< Index << ": ";
16:      cin >> *(pNumbers + Index);
17:   }
18:
19:   cout << "Wyświetlenie wszystkich podanych liczb: " << endl;
20:   for(int Index = 0; Index < InputNums; ++Index)
21:      cout << *(pCopy++) << " ";
22:
23:    cout << endl;
24:
25:    // Po zakończeniu pracy ze wskaźnikiem trzeba zwolnić pamięć.
26:    delete[] pNumbers;
27:
28:    return 0;
29: }
```

Wynik ▼

```
Ile liczb całkowitych chcesz podać? 2
Udało się zaalokować pamięć dla 2 liczb całkowitych
Podaj liczbę 0: 789
Podaj liczbę 1: 575
Wyświetlenie wszystkich podanych liczb:
789 575
```

Następne uruchomienie programu:

```
Ile liczb całkowitych chcesz podać? 5
Udało się zaalokować pamięć dla 5 liczb całkowitych
Podaj liczbę 0: 789
Podaj liczbę 1: 12
Podaj liczbę 2: -65
Podaj liczbę 3: 285
Podaj liczbę 4: -101
Wyświetlenie wszystkich podanych liczb:
789 12 -65 285 -101
```

Analiza ▼

Przed alokacją pamięci w wierszu 9. program prosi użytkownika o podanie ilości liczb całkowitych, które mają być przechowywane w systemie. Zwróć uwagę na wykonanie kopii adresu w wierszu 10. do późniejszego użycia

podczas zwalniania bloku pamięci w wierszu 26. przy użyciu operatora `delete`. W tym programie pokazano zalety użycia wskaźników i dynamicznej alokacji pamięci w porównaniu do zastosowania tablicy statycznej. Omawiana aplikacja zużywa mniejszą ilość pamięci, gdy użytkownik przechowuje małą ilość liczb całkowitych, oraz większą, kiedy trzeba przechować dużą ilość liczb całkowitych. W żadnym z wymienionych przypadków nie są marnowane zasoby systemowe. Z powodu zastosowania dynamicznej alokacji pamięci nie ma żadnej sztucznej górnej granicy ilości liczb całkowitych, które mogą być przechowywane — jedynym ograniczeniem jest dostępność zasobów systemowych. W wierszach od 13. do 17. znajduje się pętla `for` pozwalająca użytkownikowi na wprowadzenie liczb, które następnie są przechowywane w kolejnych miejscach w pamięci przy użyciu wyrażenia zdefiniowanego w wierszu 16. Do wskaźnika została dodana liczona od zera wartość przesunięcia (`Index`). Kompilator tworzy więc program wstawiający wartość podaną przez użytkownika w odpowiednim miejscu w pamięci bez nadpisywania poprzedniej. Innymi słowy, `(pNumber + Index)` to wyrażenie, które zwraca wskaźnik prowadzący do liczby całkowitej w rozpoczynającym się od zera indeksie położenia w pamięci. W takim przypadku wartość 1 indeksu wskazuje drugą liczbę całkowitą. Z kolei operator dereferencji w `*(pNumber + Index)` to wyrażenie używane przez polecenie `cin` w celu uzyskania dostępu do wartości we wspomnianym indeksie rozpoczynającym się od zera. Pętla `for` w wierszach 21. i 22. jest używana do wyświetlania wartości przechowywanych w poprzedniej pętli. Polecenie `for` wykorzystuje wiele inicjalizatorów, tworzy kopię w `pCopy` i inkrementuje tę kopię w wierszu 21., aby wyświetlić wartość.

Powodem utworzenia kopii w wierszu 10. jest fakt, że pętla z wykorzystaniem operatora inkrementacji modyfikuje (++) używany wskaźnik. Początkowy wskaźnik zwrócony przez operator `new` musi być zachowany nietknięty, aby w wierszu 26. można było wywołać odpowiednie polecenie `delete[]` wraz z dokładną wartością podaną przez operator `new`, a nie inną, przypadkową wartością.

Użycie słowa kluczowego const ze wskaźnikami

W lekcji 3. dowiedziałeś się, że deklaracja zmiennej jako `const` powoduje przypisanie tego rodzaju zmiennej wartości na stałe, tzn. wartości, która nie może być zmieniana w trakcie cyklu życiowego zmiennej. Tak więc wartość takiej zmiennej nie może być zmodyfikowana lub użyta jako l-wartość.

Wskaźniki również są zmiennymi, a tym samym słowo kluczowe `const` ma dla nich znaczenie. Jednak wskaźnik to specjalny rodzaj zmiennej zawierającej adres w pamięci i używanej do modyfikacji bloku danych w pamięci. Dlatego też dla wskaźników i stałych mamy do dyspozycji następujące rozwiązania.

- Dane wskazywane przez wskaźnik są stałą i nie mogą być zmodyfikowane, adres zawarty we wskaźniku może ulec zmianie, tzn. wskaźnik może prowadzić do zupełnie innego miejsca w pamięci.

```
int HoursInDay = 24;
const int* pInteger = &HoursInDay; // Nie można użyć pInteger do zmiany.
HoursInDay
int MonthsInYear = 12;
pInteger = &MonthsInYear; // OK!
*pInteger = 13; // Błąd kompilacji: nie można zmienić danych.
int* pAnotherPointerToInt = pInteger; // Błąd kompilacji: stałej nie można
                                      // przypisać niestałej.
```

- Adres zawarty we wskaźniku jest stałą i nie może być zmodyfikowany, ale zmienione mogą być dane, do których prowadzi wskaźnik.

```
int DaysInMonth = 30;
// Wskaźnik pInteger nie może prowadzić do niczego innego.
int* const pDaysInMonth = &DaysInMonth;
*pDaysInMonth = 31; // OK! Wartość może być zmieniona.
int DaysInLunarMonth = 28;
pDaysInMonth = &DaysInLunarMonth; // Błąd kompilacji: nie można zmienić
                                  // adresu!
```

- Zarówno adres znajdujący się we wskaźniku, jak i wartość, do której on prowadzi, są stałymi i nie mogą być zmodyfikowane (to najbardziej restrykcyjny wariant).

```
int HoursInDay = 24;
// Wskaźnik może prowadzić jedynie do HoursInDay.
const int* const pHoursInDay = &HoursInDay;
*pHoursInDay = 25; // Błąd kompilacji: nie można zmienić wartości wskazywanej
                   // przez wskaźnik.
int DaysInMonth = 30;
pHoursInDay = &DaysInMonth; // Błąd kompilacji: nie można zmienić wartości
                            // wskaźnika.
```

Wymienione różne formy `const` są szczególnie użyteczne podczas przekazywania wskaźników funkcjom. Parametry funkcji muszą być zadeklarowane w sposób zapewniający obsługę maksymalnie najwyższego (restrykcyjnego) poziomu zachowania stałości, aby zagwarantować, że funkcja (jeśli nie ma takiej potrzeby) nie będzie modyfikowała wartości,

do której prowadzi wskaźnik. Takie rozwiązanie pomaga w zachowaniu przejrzystości funkcji, zwłaszcza gdy zmiany w funkcji są wprowadzane na przestrzeni dłuższego czasu lub zmieniają się pracujący nad nią programiści.

Przekazywanie wskaźników funkcjom

Wskaźniki to efektywny sposób przekazania funkcji adresu w pamięci, zawierającego wartość, lub zwrócenia wyniku wygenerowanego przez funkcję. Podczas używania wskaźników z funkcjami bardzo ważne staje się zapewnienie, że funkcja wywołująca może modyfikować jedynie te parametry, które chcesz zmienić, a nie inne. Przykładowo funkcja obliczająca pole okręgu na podstawie danego promienia przekazywanego przez wskaźnik nie powinna mieć możliwości modyfikacji wspomnianego promienia. W takiej sytuacji użycie słowa kluczowego `const` we wskaźniku pozwala na efektywną kontrolę tego, co funkcja może modyfikować i czego nie może. Przykład rozwiązania przedstawiono w listingu 8.10.

Listing 8.10. Użycie słowa kluczowego new podczas obliczania pola okręgu, gdy promień i Pi są dostarczane w postaci stałych

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: void CalcArea(const double* const pPi, // Stała wskaźnika do stałej danych.
 4:               const double* const pRadius, // Nic nie można zmienić.
 5:               double* const pArea) // Zmiana wskazywanej wartości, a nie adresu.
 6: {
 7:    // Przed użyciem wskaźników należy je sprawdzić!
 8:    if (pPi && pRadius && pArea)
 9:       *pArea = (*pPi) * (*pRadius) * (*pRadius);
10: }
11:
12: int main()
13: {
14:    const double Pi = 3.1416;
15:
16:    cout << "Podaj promień okręgu: ";
17:    double Radius = 0;
18:    cin >> Radius;
19:
20:    double Area = 0;
21:    CalcArea (&Pi, &Radius, &Area);
22:
23:    cout << "Pole wynosi = " << Area << endl;
```

```
24:
25:    return 0;
26: }
```

Wynik ▼

```
Podaj promień okręgu: 10.5
Pole wynosi = 346.361
```

Analiza ▼

W wierszach od 3. do 5. pokazano dwie formy `const`, gdzie zarówno wskaźnik `pRadius`, jak i `pPi` zostały dostarczone w postaci „stałego wskaźnika do stałych danych". Tak więc nie ma możliwości modyfikacji adresu znajdującego się we wskaźniku oraz danych, do których ten wskaźnik prowadzi. Wskaźnik `pArea` to parametr przeznaczony do przechowywania danych wyjściowych, wartość wskaźnika (adres) nie może być modyfikowana, ale dane, do których prowadzi, mogą być zmienione. W wierszu 7. widać, jak parametry funkcji w postaci wskaźników są sprawdzane pod kątem poprawności, zanim zostaną użyte. Nie chcesz przecież, aby funkcja obliczyła pole, jeśli wywołujący przypadkowo przekaże wskaźnik `null` jako dowolny z trzech wymienionych parametrów, ponieważ to doprowadzi do awarii aplikacji.

Podobieństwa pomiędzy tablicami i wskaźnikami

Czy nie uważasz, że przykład przedstawiony w listingu 8.9, gdzie wskaźnik był inkrementowany za pomocą rozpoczynającego się od zera indeksu w celu uzyskania dostępu do kolejnej liczby całkowitej w pamięci, ma wiele wspólnego ze sposobem indeksowania tablic? Kiedy deklarujesz tablicę liczb całkowitych przy użyciu poniższego polecenia:

```
int MyNumbers[5];
```

kompilator alokuje ściśle określoną i stałą ilość pamięci do przechowywania pięciu liczb całkowitych i daje wskaźnik prowadzący do pierwszego elementu tablicy. Wskaźnik jest identyfikowany przez nazwę przypisaną zmiennej tablicy. Innymi słowy, `MyNumber` to wskaźnik prowadzący do pierwszego elementu — `MyNumber[0]`. Przedstawioną korelację zaprezentowano w listingu 8.11.

Listing 8.11. Pokazanie, że zmienna w postaci tablicy jest wskaźnikiem do pierwszego elementu

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:     // Tablica statyczna pięciu liczb całkowitych.
 6:     int MyNumbers[5];
 7:
 8:     // Tablica zostaje przypisana wskaźnikowi typu int.
 9:     int* pNumbers = MyNumbers;
10:
11:     // Wyświetlenie adresu przechowywanego przez wskaźnik.
12:     cout << "pNumbers = 0x" << hex << pNumbers << endl;
13:
14:     // Adres pierwszego elementu tablicy.
15:     cout << "&MyNumbers[0] = 0x" << hex << &MyNumbers[0] << endl;
16:
17:     return 0;
18: }
```

Wynik ▼

```
pNumbers = 0x004BFE8C
&MyNumbers[0] = 0x004BFE8C
```

Analiza ▼

W powyższym prostym programie pokazano, że zmienna tablicy może być przypisana wskaźnikowi tego samego typu (wiersz 9.). W zasadzie otrzymujemy potwierdzenie, że tablica jest zbliżona do wskaźnika. W wierszach od 12. do 15. pokazano, że adres przechowywany we wskaźniku jest taki sam jak adres w pamięci dla umieszczenia pierwszego elementu tablicy (czyli o wartości indeksu 0). Ten program jasno pokazuje, że tablica jest wskaźnikiem do pierwszego elementu znajdującego się w tablicy.

Dostęp do drugiego elementu tablicy uzyskasz za pomocą wyrażenia `MyNumbers[1]`, ale możesz również wykorzystać wskaźnik `pNumbers` i składnię `*(pNumbers + 1)`. Trzeci element tablicy statycznej jest dostępny przy użyciu wyrażenia `MyNumbers[2]`, natomiast w tablicy dynamicznej można w tym celu użyć składni `*(pNumbers + 2)`.

Ponieważ zmienne tablic to w zasadzie wskaźniki, powinno być możliwe wykorzystanie operatora dereferencji (*) używanego we wskaźnikach podczas pracy z tablicami. Podobnie powinno być możliwe użycie operatora tablicy ([]) do pracy ze wskaźnikami, co przedstawiono w listingu 8.12.

Listing 8.12. Uzyskanie dostępu do elementów tablicy za pomocą operatora dereferencji (*) i użycie operatora tablicy ([]) wraz ze wskaźnikiem

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    const int ARRAY_LEN = 5;
 6:
 7:    // Zainicjalizowana tablica statyczna pięciu liczb całkowitych.
 8:    int MyNumbers[ARRAY_LEN] = {24, -1, 365, -999, 2011};
 9:
10:    // Wskaźnik zainicjalizowany z pierwszym elementem tablicy.
11:    int* pNumbers = MyNumbers;
12:
13:    cout << "Wyświetlenie tablicy za pomocą składni wskaźnika,
       ↪operatora*" << endl;
14:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
15:       cout << "Element " << Index << " = " << *(MyNumbers + Index) <<
          ↪endl;
16:
17:    cout << "Wyświetlenie tablicy za pomocą wskaźnika wraz ze składnią
       ↪tablicy, operator[]" << endl;
18:    for (int Index = 0; Index < ARRAY_LEN; ++Index)
19:       cout << "Element " << Index << " = " << pNumbers[Index] << endl;
20:
21:    return 0;
22: }
```

Wynik ▼

```
Wyświetlenie tablicy za pomocą składni wskaźnika, operatora*
Element 0 = 24
Element 1 = -1
Element 2 = 365
Element 3 = -999
Element 4 = 2011
Wyświetlenie tablicy za pomocą wskaźnika wraz ze składnią tablicy, operator[]
Element 0 = 24
Element 1 = -1
```

```
Element 2 = 365
Element 3 = -999
Element 4 = 2011
```

Analiza ▼

Aplikacja deklaruje tablicę statyczną pięciu liczb całkowitych, która jest zainicjalizowana wraz z pięcioma wartościami (patrz wiersz 8.). Program wyświetla na ekranie zawartość tablicy, korzysta przy tym z dwóch alternatywnych rozwiązań. Pierwsze polega na użyciu zmiennej tablicy wraz z operatorem dereferencji (patrz wiersz 15.), natomiast drugie wykorzystuje zmienną wskaźnika wraz z operatorem tablicy (patrz wiersz 19.).

Działanie programu pokazuje więc, że zarówno tablica `MyNumbers`, jak i wskaźnik `pNumber` w rzeczywistości działają jak wskaźnik. Innymi słowy, deklaracja tablicy jest podobna do wskaźnika, który będzie utworzony w celu przeprowadzania operacji w ramach na stałe zdefiniowanego obszaru pamięci. Zwróć uwagę na możliwość przypisania tablicy wskaźnikowi (wiersz 11.), ale też na brak możliwości przypisania wskaźnika do tablicy — wynika to z natury tablicy statycznej, która nie może być użyta jako l-wartość.

Ostrzeżenie

Musisz koniecznie zapamiętać, że wskaźniki zaalokowane dynamicznie za pomocą operatora `new` muszą być zwalniane przy użyciu operatora `delete` nawet wtedy, kiedy skorzystałeś ze składni podobnej do składni tablicy statycznej.

Jeżeli zapomnisz o tym, w aplikacji wystąpi wyciek pamięci, a tego należy unikać za wszelką cenę.

Najczęstsze błędy programistyczne popełniane podczas używania wskaźników

Język C++ pozwala na dynamiczną alokację pamięci, co sprawia, że zużycie pamięci przez aplikację jest optymalne. W przeciwieństwie do nowszych języków programowania, takich jak C# i Java, opartych na środowisku uruchomieniowym, C++ nie oferuje mechanizmu automatycznego usuwania nieużytków odpowiedzialnego za zwalnianie pamięci zaalokowanej przez aplikację, ale nieużywanej. Ponieważ temat wskaźników jest trudny, programista ma — niestety — wiele możliwości popełnienia błędów.

Wycieki pamięci

Jest tp prawdopodobnie jeden z najczęściej spotykanych problemów z aplikacjami utworzonymi w C++: im dłużej program działa, tym większą ilość pamięci zużywa, a sam system działa coraz wolniej. Tego rodzaju sytuacja występuje zwykle wtedy, kiedy programista nie zadbał o to, by w aplikacji pamięć dynamicznie zaalokowana przy użyciu operatora `new` była zwalniania przez odpowiednie wywołanie `delete`, gdy dany blok pamięci nie jest dłużej potrzebny.

Obowiązkiem programisty jest zagwarantowanie, że cała zaalokowana przez aplikację pamięć zostanie przez nią zwolniona. Poniżej przedstawiono sytuację, która nigdy nie powinna mieć miejsca.

```
int* pNumbers = new int [5]; // Początkowa alokacja.
// Użycie wskaźnika pNumbers.
...
// Zapomnienie o konieczności użycia polecenia delete[] pNumbers; w celu zwolnienia pamięci.
...
// Przeprowadzenie innej alokacji i nadpisanie wskaźnika.
pNumbers = new int[10]; // Wyciek poprzednio zaalokowanej pamięci.
```

Kiedy wskaźnik nie prowadzi do poprawnego adresu w pamięci?

Kiedy odwołujesz się do wskaźnika za pomocą operatora dereferencji w celu uzyskania dostępu do wartości wskazywanej przez ten wskaźnik, musisz mieć 100% pewności, że wskaźnik zawiera prawidłowy adres w pamięci. W przeciwnym razie program może ulec awarii lub zachowywać się niezgodnie z oczekiwaniami. Wydaje się logiczne, że nieprawidłowe wskaźniki to również częsta przyczyna awarii aplikacji. Wskaźnik może być nieprawidłowy z wielu różnych powodów, ale wszystkie są powiązane z kiepskim zarządzaniem pamięcią. Typową sytuacją, w której wskaźnik może stać się nieprawidłowy, przedstawiono w listingu 8.13.

Listing 8.13. Przykład złego programowania z użyciem niepoprawnych wskaźników

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:     // Niezainicjalizowany wskaźnik (błąd).
6:     int* pTemperature;
7:
```

```
 8:    cout << "Czy jest pogodnie (t/n)?" << endl;
 9:    char UserInput = 't';
10:    cin >> UserInput;
11:
12:    if (UserInput == 't')
13:    {
14:       pTemperature = new int;
15:       *pTemperature = 30;
16:    }
17:
18:    // Wskaźnik pTemperature zawiera nieprawidłową wartość, jeśli użytkownik nacisnął
       ↪klawisz 'n'.
19:    cout << "Temperatura wynosi: " << *pTemperature;
20:
21:    // Wywołanie operatora delete nawet wtedy, gdy nie został użyty operator new.
22:    delete pTemperature;
23:
24:    return 0;
25: }
```

Wynik ▼

```
Czy jest pogodnie (t/n)? t
Temperatura wynosi: 30
```

Następne uruchomienie programu:

```
Czy jest pogodnie (t/n)? n
<AWARIA PROGRAMU!>
```

Analiza ▼

W powyższym programie występuje wiele problemów, niektóre z nich zostały już wskazane w kodzie. Zwróć uwagę na alokację pamięci i przypisanie jej wskaźnikowi (wiersz 14.), co następuje warunkowo po naciśnięciu klawisza *t* przez użytkownika. W przypadku naciśnięcia innego dowolnego klawisza blok `if` nie zostanie wykonany i wskaźnik `pTemperature` pozostanie nieprawidłowy. Dlatego też naciśnięcie klawisza *n* przez użytkownika podczas drugiego uruchomienia programu prowadzi do awarii aplikacji. Powód awarii to nieprawidłowy adres w pamięci przechowywany przez wskaźnik `pTemperature`, do którego odwołanie następuje w wierszu 19.

Równie niebezpieczne jest przeprowadzone w wierszu 22. wywołanie operatora `delete` względem wskaźnika, dla którego nie zaalokowano pamięci przy użyciu operatora `new`. Jeżeli masz kopię wskaźnika, wywołanie `delete`

wystarczy przeprowadzić tylko na jednej kopii (staraj się również unikać posiadania zbyt wielu kopii wskaźnika).

Lepsza (tzn. bezpieczniejsza i stabilniejsza) wersja programu przedstawionego w listingu 8.13 będzie miała zainicjalizowane wskaźniki używane po wcześniejszym sprawdzeniu poprawności oraz zwalniane jednokrotnie i tylko wtedy, gdy są poprawne.

Zawieszone wskaźniki (nazywane również zabłąkanymi)

Zauważ, że każdy prawidłowy wskaźnik staje się nieprawidłowy po jego zwolnieniu za pomocą operatora `delete`. Jeżeli wskaźnik `pTemperature` miałby być użyty po wierszu 22., nawet w sytuacji, gdy użytkownik nacisnął klawisz *t* i wskaźnik zachował ważność do tej chwili, po wywołaniu `delete` staje się nieprawidłowy i nie powinien być używany.

Aby uniknąć tego rodzaju problemu, wielu programistów przypisuje wskaźnikom wartość `null` podczas inicjalizacji lub po ich zwolnieniu, a przed użyciem wskaźnika za pomocą operatora dereferencji sprawdza jego poprawność.

Najlepsze praktyki podczas pracy ze wskaźnikami

Poniżej wymieniono pewne podstawowe reguły, których zastosowanie podczas używania wskaźników w aplikacji powinno ułatwić pracę.

TAK	NIE
Zawsze przeprowadzaj inicjalizację zmiennych wskaźników, ponieważ w przeciwnym razie będą zawierały zupełnie przypadkowe wartości. Wspomniane wartości są interpretowane jako adresy w pamięci, ale aplikacja nie ma uprawnień do uzyskania do nich dostępu. Jeżeli nie możesz zainicjalizować wskaźnika z poprawnym adresem zwróconym przez operator `new` lub inną prawidłową zmienną, zainicjalizuj go z wartością `null`.	Nie próbuj uzyskać dostępu do bloku pamięci, używając wskaźnika po jego zwolnieniu za pomocą operatora delete. Nie wywołuj więcej niż tylko jednokrotnie operatora `delete` względem adresu w pamięci.

TAK	NIE
Przed użyciem wskaźnika sprawdź, czy nie ma przypisanej wartości `null`. Tego rodzaju sprawdzenie gwarantuje, że wskaźniki nieposiadające przypisanych poprawnych adresów w poleceniach znajdujących się po ich deklaracji (gdzie zostały zainicjalizowane wraz z wartościami null) nie będą mogły być użyte (przykład takiego rozwiązania pokazano w listingu 8.13). Upewnij się o zaprogramowaniu wskaźników w taki sposób, że będą używane jedynie po potwierdzeniu ich poprawności. W przeciwnym przypadku może dojść do awarii aplikacji. Pamiętaj o użyciu operatora `delete` w celu zwolnienia pamięci zaalokowanej przez operator `new`. W przeciwnym razie aplikacja będzie miała wyciek pamięci, który może doprowadzić do zmniejszenia wydajności działania systemu.	Nie twórz wycieku pamięci w aplikacji na skutek zapomnienia o konieczności użycia operatora `delete` po zakończeniu pracy z zaalokowanym blokiem pamięci.

Po zapoznaniu się z najlepszymi praktykami podczas pracy ze wskaźnikami nadeszła doskonała chwila na poprawienie wyjątkowo nieudanego kodu przedstawionego w listingu 8.13. Poprawiona wersja kodu znajduje się w listingu 8.14.

Listing 8.14. Bezpieczniejsze programowanie wskaźników, poprawiona wersja listingu 8.13

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    cout << "Czy jest pogodnie (t/n)? ";
 6:    char UserInput = 't';
 7:    cin >> UserInput;
 8:
 9:    if (UserInput == 't')
10:    {
11:       // Zainicjalizowany wskaźnik (to dobrze).
12:       int* pTemperature = new int;
13:       *pTemperature = 30;
14:
15:       cout << "Temperatura wynosi: " << *pTemperature << endl;
```

```
16:
17:         // Użycie operatora delete po zakończeniu pracy ze wskaźnikiem.
18:         delete pTemperature;
19:     }
20:
21:     return 0;
22: }
```

Wynik ▼

```
Czy jest pogodnie (t/n)? t
Temperatura wynosi: 30
```

Następne uruchomienie programu:

```
Czy jest pogodnie (t/n)? n
```

(Zakończenie działania bez awarii).

Analiza ▼

Podstawowa różnica polega na tym, że wskaźnik jest tworzony wtedy, gdy naprawdę jest potrzebny, czyli po naciśnięciu klawisza *t* przez użytkownika, i zainicjalizowany w chwili tworzenia (patrz wiersz 12.). Zwolnienie pamięci następuje w tym samym bloku, a tym samym nie występuje sytuacja, w której wskaźnik został użyty (w odwołaniu lub w wywołaniu `delete`), kiedy nie miał przypisanego prawidłowego adresu w pamięci.

Sprawdzenie, czy żądanie alokacji zakończyło się powodzeniem

Działanie operatora `new` kończy się powodzeniem, o ile użytkownik nie zażąda alokacji ogromnej ilości pamięci lub system nie znajduje się w stanie krytycznym i ma niewielką ilość dostępnych zasobów. Istnieją aplikacje żądające dużych ilości pamięci (np. bazy danych) i dlatego ważne jest, aby nie przyjmować założenia, że alokacja pamięci na pewno zakończy się niepowodzeniem. Język C++ oferuje dwie metody zagwarantowania poprawności wskaźnika. Metoda domyślna używa wyjątków, zakończona niepowodzeniem operacja alokacji spowoduje zgłoszenie wyjątku w postaci obiektu `std::bad_alloc`. Wynikiem zgłoszenia tego wyjątku jest przerwanie działania aplikacji, tak więc jeśli nie przygotowałeś *procedury obsługi wyjątków*, aplikacja zakończy działanie wraz z komunikatem błędu informującym o nieobsłużonym wyjątku.

W lekcji 28., zatytułowanej „Obsługa wyjątków”, temat związany z obsługą wyjątków zostanie dokładnie omówiony. W listingu 8.15 znajdziesz przykład rozwiązania, w którym procedura obsługi wyjątków jest używana do sprawdzenia, czy alokacja pamięci zakończyła się niepowodzeniem.

Listing 8.15. Obsługa wyjątków, eleganckie zakończenie pracy, gdy wywołanie operatora new zakończy się niepowodzeniem

```
0: #include <iostream>
1: using namespace std;
2:
3: int main()
4: {
5:    try
6:    {
7:       // Żądanie alokacji dużej ilości pamięci.
8:       int* pAge = new int [536870911];
9:
10:       // Użycie zaalokowanej pamięci.
11:
12:       delete[] pAge;
13:    }
14:    catch (bad_alloc)
15:    {
16:       cout << "Alokacja pamięci zakończona niepowodzeniem. Zamknięcie
          ↪programu" << endl;
17:    }
18:    return 0;
19: }
```

Wynik ▼

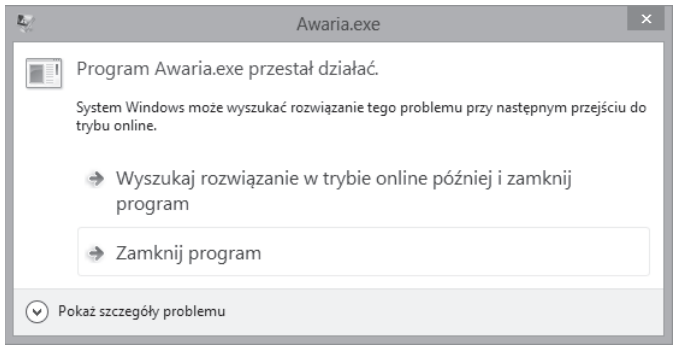
```
Alokacja pamięci zakończona niepowodzeniem. Zamknięcie programu
```

Analiza ▼

Program ten może być wykonany na różne sposoby w konkretnych komputerach. Moje środowisko pracy nie pozwala na spełnienie żądania alokacji pamięci dla 536870911 liczb całkowitych. Gdyby nie przygotowana procedura obsługi wyjątków (blok catch w wierszach od 14. do 17.), program zakończyłby niespodziewanie działanie. Uruchomienie programu w trybie debugowania w Visual Studio powoduje wygenerowanie danych wyjściowych pokazanych na rysunku 8.2.

RYSUNEK 8.2. Awaria programu na skutek braku w listingu 8.15 procedury obsługi wyjątku (program uruchomiony w trybie debugowania w kompilatorze MSVC)

Tryb debugowania powoduje wstawienie do środowiska programistycznego procedur obsługi wyjątków, a skutkiem ich działania jest komunikat pokazany na rysunku 8.2. Z kolei po uruchomieniu programu w trybie Release system operacyjny (tutaj Windows) powoduje zakończenie działania aplikacji w sposób pokazany na rysunku 8.3.



RYSUNEK 8.3. Awaria programu na skutek braku w listingu 8.15 procedury obsługi wyjątku (program uruchomiony w trybie Release)

Kiedy aplikacja ulega awarii w sposób pokazany na rysunku 8.3, jej działanie zostaje zakończone przez system operacyjny. W takim przypadku brak zaimplementowanej procedury obsługi wyjątków uniemożliwia nawet wyświetlenie jakiegoś sensownego komunikatu użytkownikowi.

Implementacja w kodzie programu procedury obsługi wyjątków daje możliwość eleganckiego zakończenia pracy aplikacji i poinformowania użytkownika o wystąpieniu problemu, zamiast pozwolenia systemowi operacyjnemu na wyświetlenie komunikatu o awarii programu.

Istnieje wariant operatora new o nazwie new(nothrow), który w przypadku nieudanej alokacji, zamiast zgłosić wyjątek, powoduje zwrócenie wskaźnikowi wartości null. Przed użyciem wskaźnika można sprawdzić jego poprawność. Wersja programu wraz z operatorem new(nothrow) została przedstawiona w listingu 8.16.

Listing 8.16. Użycie operatora new(nothrow), który zwraca wartość null, gdy alokacja zakończy się niepowodzeniem

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    // Żądanie alokacji dużej ilości pamięci, wersja, która nie zgłasza wyjątku.
 6:    int* pAge = new(nothrow) int [0x1fffffff];
 7:
 8:    if (pAge) // Sprawdzenie pAge != NULL.
 9:    {
10:       // Użycie zaalokowanej pamięci.
11:       delete[] pAge;
12:    }
13:    else
14:       cout << "Alokacja pamięci zakończona niepowodzeniem. Zamknięcie
          ↪programu" << endl;
15:
16:    return 0;
17: }
```

Wynik ▼

```
Alokacja pamięci zakończona niepowodzeniem. Zamknięcie programu
```

Analiza ▼

Program jest taki sam jak w poprzednim przykładzie (patrz listing 8.15), ale używa operatora new(nothrow), który w przypadku nieudanej alokacji pamięci zwraca wartość null, zamiast zgłosić wyjątek std::bad_alloc. Obie przedstawione formy są dobre, wybór odpowiedniej należy do Ciebie.

Czym jest referencja?

Referencja jest aliasem dla zmiennej. Po zadeklarowaniu referencji konieczne jest jej zainicjalizowanie ze zmienną. Dlatego też zmienna referencji to po prostu inny sposób uzyskania dostępu do danych we wskazanej zmiennej. Referencję deklaruje się przy użyciu operatora referencji (&), jak to przedstawiono w poniższym fragmencie kodu:

```
TypZmiennej NazwaZmiennej = Wartość;
TypZmiennej& ZmiennaReferencyjna = NazwaZmiennej;
```

Aby dokładnie zrozumieć, jak zadeklarować referencje i używać ich w kodzie, zapoznaj się z listingiem 8.17.

Listing 8.17. Pokazanie, że referencje są aliasami dla przypisanych wartości

```
 0: #include <iostream>
 1: using namespace std;
 2:
 3: int main()
 4: {
 5:    int Original = 30;
 6:    cout << "Zmienna Original = " << Original << endl;
 7:    cout << "Zmienna Original jest pod adresem: " << hex << &Original <<
       ↪endl;
 8:
 9:    int& Ref = Original;
10:    cout << "Ref jest pod adresem: " << hex << &Ref << endl;
11:
12:    int& Ref2 = Ref;
13:    cout << "Ref2 jest pod adresem: " << hex << &Ref2 << endl;
14:    cout << "Ref2 pobiera wartość, Ref2 = " << dec << Ref2 << endl;
15:
16:    return 0;
17: }
```

Wynik ▼

```
Zmienna Original = 30
Zmienna Original jest pod adresem: 0044FB5C
Ref jest pod adresem: 0044FB5C
Ref2 jest pod adresem: 0044FB5C
Ref2 pobiera wartość, Ref2 = 30
```

Analiza ▼

Dane wyjściowe programu pokazują, że referencja, niezależnie od tego, czy została zainicjalizowana z oryginalną zmienną (patrz wiersz 9.), czy do innej referencji (patrz wiersz 12.), zawsze odwołuje się do tego samego położenia w pamięci, w którym początkowo się znajdowała. Dlatego też referencje są prawdziwymi aliasami, po prostu innymi nazwami zmiennej `Original`. Wyświetlona w wierszu 14. wartość `Ref2` jest taka sama jak zmiennej `Original` (wyświetlona w wierszu 6.), ponieważ `Ref2` jest aliasem do `Original` i prowadzi do tego samego miejsca w pamięci.

Dlaczego referencje są użyteczne?

Referencje pozwalają na pracę z miejscem w pamięci, z którym zostały zainicjalizowane. Z tego powodu referencje są szczególnie użyteczne podczas programowania funkcji. Jak się dowiedziałeś w lekcji 7., zatytułowanej „Funkcje", typowa funkcja jest deklarowana w przedstawiony poniżej sposób:

```
TypZwrotny DowolnaOperacja(Typ Parametr);
```

Funkcja `DowolnaOperacja()` jest wywoływana następująco:

```
TypZwrotny Wynik = DowolnaOperacja(argument); // Wywołanie funkcji.
```

Powyższe polecenie powoduje skopiowanie `argumentu` do `Parametru`, który następnie będzie użyty przez funkcję `DowolnaOperacja()`. Krok kopiowania może być całkiem sporym obciążeniem, jeśli dany `argument` zużywa dużą ilość pamięci. Podobnie gdy funkcja `DowolnaOperacja()` zwraca wartość, jest ona kopiowana z powrotem do zmiennej `Wynik`. Byłoby idealnie, gdyby można było pominąć krok kopiowania i umożliwić funkcji bezpośrednią pracę z danymi na stosie programu wywołującego. Dzięki referencjom takie rozwiązanie jest możliwe.

Wersja funkcji bez kroku kopiowania przedstawia się następująco:

```
TypZwrotny DowolnaOperacja(Typ& Parametr); // Zwróć uwagę na referencję &.
```

Funkcję tę można wywołać w poniższy sposób:

```
TypZwrotny Wynik = DowolnaOperacja(argument);
```

Ponieważ argument jest przekazywany przez referencję, `Parametr` nie jest kopią `argumentu`, a raczej aliasem do niego, podobnie jak w przypadku `Ref` w listingu 8.17. Ponadto funkcja akceptująca parametr jako referencję może opcjonalnie zwracać wartość, używając parametru referencji. Zapoznaj się

z listingiem 8.18, aby przekonać się, jak funkcja może używać referencji zamiast wartości zwrotnej.

Listing 8.18. Funkcja obliczająca kwadrat zwrócony później w parametrze przez referencję

```
0: #include <iostream>
1: using namespace std;
2:
3: void ReturnSquare(int& Number)
4: {
5:    Number *= Number;
6: }
7:
8:
9: int main()
10: {
11:    cout << "Podaj liczbę, którą chcesz podnieść do kwadratu: ";
12:    int Number = 0;
13:    cin >> Number;
14:
15:    ReturnSquare(Number);
16:    cout << "Kwadrat podanej liczby wynosi: " << Number << endl;
17:
18:    return 0;
19: }
```

Wynik ▼

```
Podaj liczbę, którą chcesz podnieść do kwadratu: 5
Kwadrat podanej liczby wynosi: 25
```

Analiza ▼

Funkcja obliczająca kwadrat podanej liczby znajduje się w wierszach od 3. do 6. Zwróć uwagę, że dane wejściowe akceptuje w postaci parametru przekazywanego przez referencję, a wynik zwraca w ten sam sposób. Jeśli zapomnisz oznaczyć parametr `Number` jako referencję (&), wynik nie zostanie przekazany wywołującej funkcji `main()`, ponieważ funkcja `ReturnSquare()` będzie przeprowadzała operację na lokalnej kopii `Number` niszczonej w chwili zakończenia działania funkcji. Dzięki użyciu referencji masz pewność, że funkcja `ReturnSquare()` działa względem tego samego adresu w pamięci, który zaalokowano dla `Number` w funkcji `main()`. Wynik operacji jest więc dostępny w funkcji `main()` nawet po zakończeniu działania funkcji `ReturnSquare()`.

W omówionym programie został zmodyfikowany parametr danych wejściowych zawierający liczbę podaną przez użytkownika. Jeżeli potrzebujesz obu wartości, czyli liczby początkowej i podniesionej do kwadratu, konieczne jest przygotowanie funkcji akceptującej dwa parametry: z liczbą początkową oraz z liczbą podniesioną do kwadratu.

Użycie słowa kluczowego const w referencjach

Być może wystąpi konieczność posiadania referencji, jaka nie będzie mogła zmienić wartości początkowej zmiennej, której jest aliasem. Użycie słowa kluczowego `const` podczas deklarowania tego rodzaju referencji to jedno z możliwych rozwiązań:

```
int Original = 30;
const int& ConstRef = Original;
ConstRef = 40; // Niedozwolone: ConstRef nie może zmienić wartości Original.
int& Ref2 = ConstRef; // Niedozwolone: Ref2 nie jest stałą.
const int& ConstRef2 = ConstRef; // OK.
```

Przekazywanie funkcji argumentów przez referencję

Jedną z największych zalet referencji jest to, że umożliwia wywoływanej funkcji pracę z parametrami, które nie muszą być kopiowane z funkcji wywołującej, co znacznie poprawia wydajność działania wywoływanej funkcji. Jednak wywoływana funkcja działa, używając parametrów bezpośrednio znajdujących się na stosie funkcji wywołującej. Bardzo często ważne jest zagwarantowanie, że wywołana funkcja nie będzie mogła zmienić wartości zmiennej w funkcji wywołującej. Referencje zdefiniowane z użyciem słowa kluczowego `const` doskonale się przydają w takich sytuacjach, co zostało zademonstrowane w listingu 8.19. Parametr referencji `const` nie może być używany jako l-wartość, więc próba przypisania mu wartości spowoduje wygenerowanie błędu w trakcie kompilacji.

Listing 8.19. Użycie referencji const w celu zagwarantowania, że wywołująca funkcja nie będzie mogła zmodyfikować wartości podanej przez referencję

```
0: #include <iostream>
1: using namespace std;
2:
```

```
 3: void CalculateSquare(const int& Number, int& Result) // Zwróć uwagę na "const".
 4: {
 5:    Result = Number*Number;
 6: }
 7:
 8: int main()
 9: {
10:    cout << "Podaj liczbę, którą chcesz podnieść do kwadratu: ";
11:    int Number = 0;
12:    cin >> Number;
13:
14:    int Square = 0;
15:    CalculateSquare(Number, Square);
16:    cout << Number << "^2 = " << Square << endl;
17:
18:    return 0;
19: }
```

Wynik ▼

```
Podaj liczbę, którą chcesz podnieść do kwadratu: 27
27^2 = 729
```

Analiza ▼

W przeciwieństwie do poprzedniego programu, w którym zmienna zawierająca liczbę początkową przechowywała także wynik działania funkcji, powyższy program używa dwóch zmiennych: jednej przeznaczonej dla liczby początkowej podanej przez użytkownika i drugiej dla wyniku przeprowadzonej operacji. Aby zagwarantować, że liczba podana przez użytkownika nie będzie mogła być zmieniona, użyto referencji `const` zdefiniowanej za pomocą słowa kluczowego `const` (patrz wiersz 3.). W ten sposób parametr `Number` jest automatycznie parametrem danych wejściowych, a jego wartość nie może być zmodyfikowana.

Jako eksperyment możesz wprowadzić modyfikację w wierszu 5., aby zwrócić wynik z użyciem logiki takiej samej jak w listingu 8.18:

```
Number *= Number;
```

Bez wątpienia w trakcie kompilacji nastąpi wygenerowanie błędu informującego o braku możliwości zmiany wartości `const`. Dlatego też referencja `const` to użyteczne narzędzie udostępniane przez język C++ wskazujące, że dany parametr jest parametrem danych wejściowych. Jednocześnie masz gwarancję, że wartość przekazywana przez referencję nie będzie mogła być modyfikowana przez

wywołaną funkcję. To może wydawać się oczywiste, ale w środowisku pracy obejmującym wielu programistów, gdzie jedna osoba tworzy pierwszą wersję funkcji, a inna poprawia ją i usprawnia, użycie referencji `const` wprowadza dużą różnicę w jakości tworzonego kodu.

Podsumowanie

W tej lekcji poznałeś wskaźniki i referencje. Dowiedziałeś się, jak wskaźniki mogą być wykorzystywane do uzyskania dostępu i operowania pamięcią oraz jakie to przydatne narzędzie pomagające w trakcie dynamicznej alokacji pamięci. Poznałeś operatory `new` i `delete` używane do alokacji pamięci dla elementu. Ponadto poznałeś ich odpowiedniki `new[...]` i `delete[]` pomagające w alokacji pamięci dla tablicy danych. W lekcji przedstawiono również pułapki czyhające na programistów w trakcie pracy ze wskaźnikami i dynamiczną alokacją pamięci. Przekonałeś się też, jak ważne jest zwalnianie dynamicznie alokowanej pamięci, aby uniknąć tzw. wycieków pamięci. Referencje są aliasami i jednocześnie oferującą potężne możliwości alternatywą dla używania wskaźników podczas przekazywania argumentów funkcjom — referencje gwarantują zachowanie ich poprawności. Dowiedziałeś się, jak we właściwy sposób stosować `const` podczas pracy ze wskaźnikami i referencjami oraz jak deklarować funkcje na najwyższym możliwym poziomie zachowania stałości parametrów.

Pytania i odpowiedzi

Pytanie: Dlaczego stosować dynamiczną alokację pamięci, skoro mogę zdefiniować tablicę statyczną i nie przejmować się zwalnianiem pamięci?

Odpowiedź: Tablice statyczne mają stałą wielkość i nigdy nie zapewniają możliwości skalowania w górę, jeśli aplikacja będzie wymagała większej ilości pamięci, ani nie optymalizują użycia pamięci, gdy aplikacja korzysta z mniejszej ilości danych. W takich sytuacjach stosowanie dynamicznej alokacji pamięci powoduje istotną różnicę.

Pytanie: Mam dwa przedstawione poniżej wskaźniki:

```
int* pNumber = new int;
int* pCopy = pNumber;
```

Czy dobrym rozwiązaniem będzie wywołanie operatora `delete` względem obu, aby zagwarantować całkowite zwolnienie pamięci?

Odpowiedź: To jest błędne rozwiązanie. Można tylko jednokrotnie wywołać operator `delete` względem danego adresu w pamięci zwróconego przez operator `new`. Ponadto warto unikać sytuacji, w której dwa wskaźniki prowadzą do tego samego adresu w pamięci, ponieważ wywołanie operatora `delete` spowoduje unieważnienie obu wskaźników. Nie powinieneś także tworzyć programu w taki sposób, że nie masz pewności dotyczącej poprawności używanych wskaźników.

Pytanie: Kiedy powinienem używać operatora `new(nothrow)`?

Odpowiedź: Jeśli nie chcesz zajmować się obsługą wyjątku `std::bad_alloc`, wtedy możesz użyć wersji `new(nothrow)` operatora `new`, ponieważ w przypadku nieudanej operacji alokacji pamięci zwraca on wartość `null`.

Pytanie: Potrzebuję wywołać funkcję obliczającą pole. Do dyspozycji mam dwie przedstawione poniżej metody:

```
void CalculateArea (const double* const pRadius, double* const pArea);
void CalculateArea (const double& radius, double& area);
```

Który z powyższych wariantów powinienem wybrać?

Odpowiedź: Drugi z wymienionych, użycie referencji jako referencji nie jest błędem, podczas gdy w przypadku wskaźników już tak. Ponadto druga metoda jest prostsza.

Pytanie: Mam dwa wskaźniki:

```
int Number = 30;
const int* pNumber = &Number;
```

Rozumiemwzględu na deklarację `const`. Czy mogę przypisać `pNumber` do wskaźnika typu innego niż `const`, a następnie użyć go do modyfikacji wartości znajdującej się w `Number`?

Odpowiedź: Nie, nie ma możliwości zmiany wskaźnika `const`:

```
int* pAnother = pNumber; // Wskaźnika const nie można przypisać do nieconst.
```

Pytanie: Dlaczego mam trudzić się przekazywaniem funkcji wartości przez referencję?

Odpowiedź: Nie musisz tego robić, o ile takie rozwiązanie nie ma dużego wpływu na kod. Jeśli jednak parametry funkcji akceptują obiekty, które mogą być całkiem duże (wielkość wyrażona w bajtach), ich przekazanie przez wartość stanie się naprawdę kosztowną operacją. Wywołanie funkcji może być znacznie efektywniejsze po użyciu referencji. Pamiętaj o częstym użyciu `const`, z wyjątkiem funkcji, które muszą w zmiennej przechowywać wynik działania.

Pytanie: Jaka jest różnica pomiędzy dwoma poniższymi deklaracjami?

```
int MyNumbers[100];
int* MyArrays[100];
```

Odpowiedź: `MyNumbers` to tablica liczb całkowitych, tzn. `MyNumbers` jest wskaźnikiem do miejsca w pamięci przechowującego sto liczb całkowitych i wskazuje pierwszą z nich (indeks 0). To jest statyczna alternatywa dla poniższego fragmentu kodu:

```
int* MyNumbers = new int [100]; // Tablica zaalokowana dynamicznie.
// Użycie MyNumbers.
delete MyNumbers;
```

Z drugiej strony, `MyArrays` to tablica stu liczb całkowitych, każdy wskaźnik może prowadzić do liczby całkowitej lub tablicy liczb całkowitych.

Warsztaty

Warsztaty zawierają pytania w formie quizu, które pomogą Ci w utrwaleniu wiedzy przedstawionej w tej lekcji, a także ćwiczenia pozwalające na sprawdzenie stopnia opanowania materiału. Spróbuj odpowiedzieć na pytania i rozwiązać quiz przed sprawdzeniem odpowiedzi w dodatku D. Zanim przejdziesz do kolejnej lekcji, upewnij się także, że rozumiesz odpowiedzi.

Quiz

1. Dlaczego nie można przypisać referencji `const` do referencji innej niż `const`?
2. Czy `new` i `delete` to funkcje?
3. Jaka jest natura wartości przechowywanej w zmiennej wskaźnika?
4. Jakiego operatora użyjesz w celu uzyskania dostępu do danych wskazywanych przez wskaźnik?

Ćwiczenia

1. Co zostanie wyświetlone po wykonaniu przedstawionych poniżej poleceń?

```
0: int Number = 3;
1: int* pNum1 = &Number;
2: *pNum1 = 20;
3: int* pNum2 = pNum1;
4: Number *= 2;
5: cout << *pNum2;
```

2. Jakie są podobieństwa i różnice pomiędzy przedstawionymi poniżej trzema przeciążonymi funkcjami?

```
int DoSomething(int Num1, int Num2);
int DoSomething(int& Num1, int& Num2);
int DoSomething(int* pNum1, int* pNum2);
```

3. Jak zmienisz deklarację pNum1 w ćwiczeniu 1. (wiersz 1.), aby przypisanie w wierszu 3. stało się nieprawidłowe? (Podpowiedź: musisz upewnić się, że pNum1 nie może zmienić wskazywanych danych).

4. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
#include <iostream>
using namespace std;
int main()
{
   int *pNumber = new int;
   pNumber = 9;
   cout << "Wartość wskaźnika pNumber: " << *pNumber;
   delete pNumber;
   return 0;
}
```

5. **Łowcy błędów:** Co jest nie tak z poniższym fragmentem kodu?

```
#include <iostream>
using namespace std;
int main()
{
   int pNumber = new int;
   int* pNumberCopy = pNumber;
   *pNumberCopy = 30;
   cout << *pNumber;
   delete pNumberCopy;
   delete pNumber;
   return 0;
}
```

6. Jakie będą dane wyjściowe powyższego programu po jego poprawieniu?

Skorowidz

A

B

C

D

E

F

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

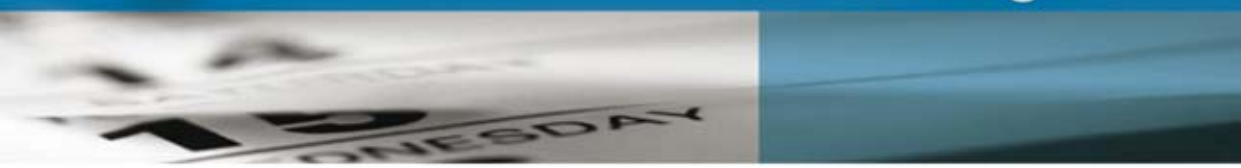
V

W

Z

C++

Dla **każdego**



Poznaj i wykorzystaj potencjał języka C++!

Pomimo swojego wieku C++ wciąż zajmuje wysoką lokatę na liście najpopularniejszych języków programowania. Przez lata dzięki wydajności oraz dostępności zdobył sobie licznych wielbicieli i wciąż jest niezastąpiony w wielu zadaniach, choć w branży przewagę nad nim mają język Java i platforma .NET. Jeżeli jednak oczekujesz najwyższej wydajności i przewidywalnego czasu wykonania zadania, język C++ może okazać się jedynym słusznym wyborem.

Z tą książką opanujesz język C++ bez trudu, poświęcając na to tylko godzinę dziennie! W trakcie lektury zgłębisz jego tajniki, poznasz zalety oraz wady. Każda kolejna godzina to coraz bardziej zaawansowana, a przy tym ciekawsza dawka wiedzy. Opanowanie materiału zawartego w podręczniku pozwoli Ci na pisanie programów o różnym stopniu złożoności oraz swobodne poruszanie się w świecie języka C++. W trakcie lektury opanujesz składnię i elementy języka — tablice, instrukcje warunkowe, pętle czy stałe nie będą miały przed Tobą żadnych tajemnic. Z kolejnych rozdziałów dowiesz się, jak wykorzystać wskaźniki i dyrektywy kompilatora. Ponadto zaznajomisz się z zasadami programowania obiektowego oraz poznasz nowości zawarte w wersji 11. Książka ta jest doskonałym źródłem wiedzy dla każdego adepta języka C++!

Dzięki tej książce:

- poznasz składnię języka C++
- opanujesz nowości wersji 11 języka C++
- poznasz zasady programowania obiektowego
- napiszesz program o dowolnym stopniu skomplikowania

helion.pl
księgarnia internetowa

Nr katalogowy: 16326

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

SAMS
Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ

KOD KORZYŚCI

ISBN 978-83-246-8077-1

9 788324 680771

cena: 99,00 zł