Politechnika Wrocławska

Wydział Elektroniki,
Fotoniki i Mikrosystemów

Laboratorium informatyki

Ćwiczenie nr 8. Referencje i dynamiczna alokacja pamięci

Zagadnienia do opracowania:

- właściwości zmiennych referencyjnych
- porównanie referencji i wskaźników
- zalety i ograniczenia stosowania referencji
- porównanie stosu i sterty
- dynamiczna alokacja pamięci w języku *C*
- dynamiczna alokacja pamięci w języku *C++*

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z alternatywnym do wskaźników mechanizmem pracy na oryginałach zmiennych, jakim są zmienne referencyjne, a także opanowanie umiejętności dynamicznego zarządzania pamięcią w językach *C* oraz *C++*.

2. Wprowadzenie

2.1. Referencje

Język *C++* wprowadził wygodną alternatywę dla wskaźników umożliwiającą pracę na oryginałach zmiennych. Stanowią je ***zmienne referencyjne***, zwane krócej ***referencjami***. Są to ***aliasy*** wcześniej zdeklarowanych już zmiennych. Oznacza to, że odwołując się do ***zmiennej referencyjnej*** odwołujemy się w rzeczywistości do innej, skojarzonej z nią zmiennej, stosując inny identyfikator (nazwę). Szczególne zastosowanie ***referencje*** znalazły jako argumenty funkcji. ***Typy referencyjne*** to *typy złożone* deklarowane z wykorzystaniem symbolu **&**, co przedstawiono na listingu 1. – **int &** stanowi ***referencję na typ int***. Referencjami możemy posługiwać się tak, jak zwykłymi zmiennymi. Modyfikując wartość referencji modyfikujemy jednocześnie wartość skojarzonej z nią zmiennej.

```
1 int value = 2;
2 // referencja do zmiennej value
3 int & refValue = value;
4
5 // Wypisze wartosc 2
6 std::cout << value << std::endl;
7 std::cout << refValue << std::endl;
8
```

```
// Modyfikacja wartosci zmiennej value z
   wykorzystaniem referencji
refValue = 5;

// Wypisze wartosc 5
std::cout << value << std::endl;
std::cout << refValue << std::endl;
```

Listing 1. Deklaracja zmiennej referencyjnej

Zmienna referencyjna i skojarzona z nią zmienna **mają ten sam adres pamięci**, co świadczy o tym, że **referencja to inna nazwa danej zmiennej**:

```
int value = 2;
int & refValue = value;

// Wypisze te sama wartosc adresu
std::cout << &value << std::endl;
std::cout << &refValue << std::endl;
```

Warto zwrócić szczególną uwagę na wykorzystanie w tym miejscu symbolu **&**, używanego również jako ***operator adresu***. Dla początkujących programistów może wydawać się nieintuicyjne kiedy symbol & określa ***referencję***, a kiedy stanowi ***operator adresu*** (analogicznie kiedy symbol * definiuje wskaźnik, a kiedy określa operator dereferencji). Reguła stanowi, że jeżeli symbol & (lub *) **stoi za typem zmiennej**, to jest to ***referencja*** (analogicznie wskaźnik) na ten typ. Taka sytuacja występuje w przypadku deklarowania bądź rzutowania zmiennych. Jeżeli symbol & (lub *) **stoi przed identyfikatorem (nazwą) zmiennej** i **nie jest bezpośrednio poprzedzony typem zmiennej**, to jest to ***operator adresu*** (analogicznie operator dereferencji). Przykłady zaprezentowano na listingu 2.

```
// int & to referencja (y jest referencja x)
int & y = x;

// int * to wskaznik (ptr jest wskaznikiem do val)
// &val to operator adresu na zmiennej val
int * ptr = &val;

// *ptr to operator dereferencji na wskazniku ptr
std::cout << *ptr << std::endl;

// *ptr to operator dereferencji na wskazniku ptr
// symbol * nie jest bezposrednio poprzedzony typem
   zmiennej - wystepuje nawias
std::cout <<  (float)*ptr << std::endl;

// long * to wskaznik (rzutowanie typow)
// identyfikator zmiennej nie jest bezposrednio
   poprzedzony symbolem * - wystepuje nawias
long * ptrCast = (long *)ptr;
```

Listing 2. Różne znaczenia symboli & i *

Kluczową cechą referencji jest **konieczność ich inicjalizacji**. Nie można zdeklarować niezainicjalizowanej zmiennej referencyjnej, aby później skojarzyć z nią innej zmiennej:

```
float val = 3.5;
float & ref; // Blad kompilacji
ref = val;
```

Analogiczne zachowanie otrzymamy deklarując wskaźnik z kwalifikatorem ***const***:

```
float val = 3.5;
float * const ptr; // Blad kompilacji
ptr = &val; // Blad kompilacji
```

Co więcej, **nie jest możliwe powiązanie referencji z inną zmienną, niż określoną w chwili inicjalizacji**:

```
long x = 123;
long & ref = x;

long y = 654;
// Przypisanie ref (oraz x) wartosci 654
ref = y;
```

W przypadku wskaźnika zdeklarowanego z użyciem kwalifikatora ***const*** taka operacja zakończy się błędem kompilacji (*assignment of read-only variable 'ptr'*):

```
long x = 123;
long * const ptr = &x;

long y = 654;
ptr = &y; // Blad kompilacji
```

Cecha ta stanowi wyraźną zaletę stosowania ***referencji*** zamiast ***wskaźników* w przypadku pracy na oryginałach zmiennych**. Niezainicjalizowany lokalny wskaźnik przyjmuje **losową wartość**, czego konsekwencje zostały omówione w Ćw. 6. Co więcej, przyjmując wskaźnik jako argument funkcji należy sprawdzić czy jest zainicjalizowany poprawną wartością (róż-

ną od ***NULL/nullptr***). Referencje nie wymagają takich zabiegów, ponieważ **wymuszają inicjalizację poprawną wartością w momencie ich deklaracji**. Należy mieć na uwadze, że referencje nie mogą zastąpić wskaźników w każdym kontekście. **Zmienne referencyjne, w przeciwieństwie do wskaźników, nie umożliwiają dynamicznej alokacji pamięci.**

Najpopularniejszym zastosowaniem ***referencji*** jest wykorzystanie ich jako argumentów funkcji. Umożliwia to przyjmowanie przez funkcje oryginałów zmiennych zamiast ich kopii (domyślny mechanizm). Inicjalizacja referencji odbywa się na etapie wywołania funkcji z określonymi argumentami. Przykład przedstawiono na listingu 3.

```
1  #include <iostream>
2  // Praca na oryginale zmiennej
3  void incrementVal(int & valRef) {
4      // Modyfikacja oryginalu
5      ++valRef;
6  }
7
8  int main() {
9      int val = 111;
10     // inicjalizacja zmiennej valRef i modyfikacja
       zmiennej val
11     incrementVal(val);
12     // Wyswiela 112
13     std::cout << val << std::endl;
14     return 0;
15 }
```

Listing 3. Praca na oryginale zmiennej

Referencje mogą być stosowane jako argumenty funkcji również, jeżeli dążymy do optymalizacji kodu (np. uniknięcia zbędnego kopiowania zmien-

nych), szczególnie w przypadku zmiennych o znacznym rozmiarze. Jednakże nie zawsze pożądana jest jednoczesna modyfikacja oryginałów zmiennych. Przed nieintencjonalną modyfikacją ***zmiennej referencyjnej*** można się zabezpieczyć stosując kwalifikator ***const***. Co więcej, funkcje przyjmujące jako argument ***stałą referencję*** mogą być wywoływane z wartościami stałymi i zmiennymi, natomiast funkcje, które przyjmują jako argument ***referencję*** nieoznaczoną kwalifikatorem ***const*** mogą być wywoływane wyłącznie ze zmiennymi nieoznaczonymi kwalifikatorem ***const***. Przykład przedstawiono na listingu 4.

```
1  void useReference(float & ref) {
2      ref = 2.0;
3  }
4
5  void useConstReference(const float & ref) {
6      // Blad kompilacji - modyfikacja stalej
7      ref = 2.0;
8  }
9
10 int main() {
11     float val = 5.5;
12     const float constVal = 5.5;
13     useConstReference(val);
14     useConstReference(constVal);
15     useReference(val);
16     // Blad kompilacji - funkcja moglaby
       zmodyfikowac stala
17     useReference(constVal);
18     return 0;
19 }
```

Listing 4. Stała referencja jako argument funkcji

Stałe referencje mają jeszcze jeden istotny aspekt. Możliwe jest wywoływanie funkcji przyjmujących ***stałe referencje*** (w przeciwieństwie do funkcji przyjmujących niestałe referencje) ze ***zmiennymi tymczasowymi***. Kompilator tworzy (anonimowe) ***zmienne tymczasowe*** o czasie życia równym czasowi obsługi funkcji. Dzieje się to w dwóch przypadkach:

- jeżeli argument wywołania ma typ niezgodny z nagłówkiem funkcji, ale jest możliwa jego niejawna konwersja na typ zgodny;
- jeżeli argument wywołania ma zgodny typ, ale nie jest ***l–wartością*** (*ang. l–value*)[1].

L–wartości to zmienne posiadające adres, do których można się odwołać (do których można zastosować operator przypisania – **operator=**) i wykorzystać wielokrotnie. Zastosowanie ***stałej referencji*** jako argumentu funkcji gwarantuje, że nie będzie możliwe zastosowanie operatora przypisania na zmiennej tymczasowej, a więc jej modyfikacja. Przykład przedstawiono na listingu 5. Usunięcie kwalifikatora ***const*** z nagłówka funkcji ***factorial()*** poskutkowałoby błędem kompilacji przy próbie wywołania funkcji ze zmienną tymczasową.

```
1  #include <iostream>
2
3  long factorial(const long & n) {
4      if (n == 0) return 1;
5      else return n * factorial(n - 1);
6  }
7
8  int main() {
9      // Zmienna tymczasowa (r-value) - brak
       mozliwosci odwolania sie, pobrania adresu
10     std::cout << factorial(3) << std::endl;
11
```

```
12      int val = 5;
13      // Zmienna tymczasowa - konwersja typu
14      std::cout << factorial(val) << std::endl;
15      return 0;
16  }
```

Listing 5. Inicjalizacja stałej referencji zmienną tymczasową

Przedstawione przykłady wskazują, że **zawsze, kiedy jest to możliwe, należy stosować kwalifikator *const* razem z referencjami**, ponieważ:

- stanowi to zabezpieczenie przed niecelową modyfikacją oryginału;
- pozwala to na wywoływanie funkcji z użyciem wyrażeń stałych i zmiennych;
- umożliwia to wywoływanie funkcji z użyciem zmiennych tymczasowych.[1]

Dobrym zwyczajem jest stosowanie kwalifikatora ***const* zawsze, kiedy jest to możliwe** (nie tylko w przypadku referencji i wskaźników) w celu optymalizacji kodu i zmniejszenia jego podatności na błędy.

Referencje można stosować również jako wartości zwracane z funkcji. Należy jednak pamiętać, tak jak w przypadku ***wskaźników***, aby **nie zwracać referencji do zmiennych lokalnych**. W przeciwnym wypadku zwrócona ***referencja*** będzie skojarzona ze skasowaną zmienną (zmienną, dla której została zwolniona pamięć). Taką zmienną referencyjną zwykło nazywać się ***dyndającą referencją*** (*ang. dangling reference*):

```cpp
#include <iostream>

int & getReference() {
    // Zmienna lokalna
    int var = 10;
    return var;
}

int main() {
    // Dyndajaca referencja
    int & ref = getReference();
    // Niezdefiniowane zachowanie!
    std::cout << ref << std::endl;
    return 0;
}
```

2.2. Stos i sterta

Stos (*ang. stack*) to segment pamięci zrealizowany w postaci ***liniowej struktury danych***, w której elementy dodawane i usuwane są z wierzchołka stosu. Jest to zatem ***bufor LIFO*** (*ang. Last In, First Out*) – ostatni element na wejściu jest pierwszym elementem na wyjściu. Na ***stosie*** odkładane są:

- zmienne lokalne alokowane statycznie (*ale nie zmienne statyczne!*),
- adresy powrotu z funkcji,
- kopie argumentów wywołania funkcji (w tym argumenty funkcji *main()*),
- wartości zwracane z funkcji.

Rozmiar pamięci ***stosu*** jest znany na etapie kompilacji, a zmienne na nim alokowane są zwalniane automatycznie, bez ingerencji użytkownika (programisty).

Sterta (*ang. heap*) to segment pamięci przechowujący niepowiązane ze sobą elementy (w sposób nieuporządkowany, możliwa fragmentacja pamięci), do których można odwołać się z każdego miejsca w programie, jeżeli zna się ich adresy. Na ***stercie*** przechowywane są zmienne zaalokowane dynamicznie, stąd rozmiar pamięci alokowanej na ***stercie*** przez program nie jest znany na etapie jego kompilacji (co nie wyklucza zjawiska ***przepełnienia sterty*** (*ang. heap overflow*)). Konstrukcja ***sterty*** powoduje, że operacje na zmiennych zaalokowanych na niej są z reguły **wolniejsze** niż w przypadku zmiennych zaalokowanych na ***stosie***. Do alokacji pamięci na ***stercie*** wykorzystuje się ***wskaźniki*** oraz dedykowane funkcje/operatory. **Pamięć zaalokowana na stercie nie jest zwalniana w sposób automatyczny. Konieczne jest do tego jawne wywołanie odpowiednich funkcji/operatorów przez programistę.** ***Sterta*** nie jest czyszczona w sposób automatyczny, między wywołaniami funkcji, jak ma to miejsce w przypadku ***stosu***. Niektóre języki programowania wysokiego poziomu zapewniają mechanizm zwany ***odśmiecaczem*** (***ang. garbage collector***), który co jakiś czas zwalnia nieużywane już zmienne zaalokowane na ***stercie*** (np. *Java* lub *Python*). Zarówno język *C*, jak i *C++*, **nie zapewniają takiego mechanizmu**. Dlatego niezwolnienie pamięci zaalokowanej na ***stercie*** prowadzi do ***wycieków pamięci*** (*ang. memory leak*) – program traktuje fragment pamięci jako zajęty i nie dopuszcza do jego ponownego użycia.

2.3. Dynamiczna alokacja pamięci

Dynamiczna alokacja pamięci polega na alokowaniu pamięci na ***stercie*** z wykorzystaniem dedykowanych funkcji bądź operatorów. Rozmiar alokowanej pamięci nie musi być znany na etapie kompilacji programu. Przeciwieństwem jest statyczna alokacja pamięci w czasie kompilacji. Dynamiczna alokacja pamięci stosowana jest gdy:

- rozmiar wymaganej pamięci nie jest znany na etapie kompilacji programu, np. rozmiar alokowanej tablicy jest zadawany przez użytkownika za pomocą *funkcji wejścia*;
- wykorzystywane struktury danych (np. tablice) mogą zwiększać swój rozmiar w trakcie działania programu (optymalizacja);
- czas życia zmiennych jest dłuższy niż czas wykonania bloku (funkcji), ale krótszy niż czas działania aplikacji (zmienne globalne), np. gdy zmienna zdefiniowana wewnątrz funkcji ma być dostępna dla innych funkcji;
- implementuje się struktury danych takie, jak *listy* (*ang. linked lists*). [*Uwaga: więcej informacji o strukturach danych w Ćw. 10*]

2.3.1. Język *C*

W języku *C* dynamiczna alokacja pamięci realizowana jest za pomocą jednej z trzech funkcji: ***malloc()***, ***calloc()*** lub ***realloc()***, natomiast zwalnianie pamięci zaalokowanej na ***stercie*** odbywa się zawsze z wykorzystaniem funkcji ***free()***. Wszystkie funkcje zostały zdeklarowane w nagłówku ***stdlib.h***; ich nagłówki zestawiono na listingu 6. Żadnej z wymienionych funkcji **nie można przeciążać**. Typ ***size_t*** to alias dla jednego z podstawowych typów całkowitych bez znaku; za jego pomocą można wyrazić rozmiar dowolnej zmiennej (w bajtach). Typ ***void ** ** to typ zmiennej wskaźnikowej niepowiązany z żadnym konkretnym typem zmiennej. Innymi słowy, zmienna typu ***void ** ** może **przechowywać adres zmiennej dowolnego typu** i może

być **rzutowana na dowolny typ wskaźnikowy**. Dzięki temu możliwa jest ***generyczna*** (niezależna od typu) implementacja funkcji realizujących dynamiczną alokację pamięci.

```
void * malloc(size_t size);
void * calloc(size_t num, size_t size);
void * realloc(void * ptr, size_t new_size);
void free(void * ptr);
```

Listing 6. Nagłówki funkcji dedykowanych do dynamicznej alokacji/zwalniania pamięci w języku *C*

Funkcja ***malloc()*** alokuje na ***stercie*** **niezainicjalizowaną** pamięć o rozmiarze ***size*** (w bajtach). W przypadku powodzenia zwracany jest **wskaźnik na początek zaalokowanego bloku pamięci**. W innym wypadku zwracana jest wartość ***NULL***. Rezultat wywołania funkcji ***malloc(0)*** jest zależny od implementacji kompilatora. Przykład dynamicznej alokacji pamięci z wykorzystaniem funkcji ***malloc()*** przedstawiono na listingu 7. Należy mieć na uwadze, że odwołanie się do obszaru pamięci ***sterty*** leżącego poza zaalokowanym segmentem skutkuje ***niezdefiniowanym zachowaniem***. [*Uwaga: w języku C można pominąć rzutowanie* ***void **** *na* ***int ****. *W języku C++ pominięcie rzutowania zakończy się błędem kompilacji.*]

```
const unsigned int tabSize = 5;
// Dynamiczna alokacja 5-elementowej tablicy liczb
   calkowitych
// Alokowany rozmiar stanowi iloczyn tabSize i
   rozmiaru pojedynczego elementu tablicy tab (int)
// Analogicznie: malloc(tabSize * sizeof(int))
int * tab = (int *) malloc(tabSize * sizeof(*tab));
if (tab != NULL) {
    // Inicjalizacja i wypisanie elementow tablicy
```

```
8       for (unsigned int i = 0; i < tabSize; ++i)
9           printf("tab[%d]=%d\n", i, tab[i] = i);
10  }
11  // Zwolnienie pamieci
12  free(tab);
```

Listing 7. Dynamiczna alokacja pamięci w języku *C*

Funkcja ***calloc()*** działa podobnie do funkcji ***malloc()***, tj. alokuje na ***stercie*** pamięć dla tablicy ***num*** elementów o rozmiarze ***size*** każdy, z tą różnicą, że **wszystkie bajty zaalokowanego obszaru pamięci są wstępnie zerowane**:

```
1   // Dynamiczna alokacja 5-elementowej tablicy liczb
        zmiennoprzecinkowych
2   const unsigned int tabSize = 5;
3   float * tab = (float *) calloc(tabSize, sizeof(*tab)
        );
4   if (tab != NULL) {
5       // Wypisanie elementow tablicy
6       for (unsigned int i = 0; i < tabSize; ++i)
7           printf("tab[%d]=%f\n", i, tab[i]);
8   }
9   // Zwolnienie pamieci
10  free(tab);
```

Funkcja ***realloc()*** alokuje segment pamięci o rozmiarze ***new_size*** w obszarze wskazywanym przez wskaźnik ***ptr***, który został wcześniej zaalokowany przez wywołanie jednej z funkcji: ***malloc()***, ***calloc()*** bądź ***realloc()*** i nie został zwolniony z użyciem funkcji ***free()***. W ten sposób możliwe jest rozszerzanie bądź zmniejszanie zaalokowanego obszaru pamięci. Jeżeli ***new_size*** jest równe 0, to rezultat wywołania funkcji jest zależny od implementacji kompilatora. Należy mieć na uwadze, że funkcja ***realloc()*** w pierwszej ko-

lejności zwalnia pamięć wskazywaną przez wskaźnik ***ptr***. Przykład realokacji pamięci przedstawiono na listingu 8.

```
// Dynamiczna alokacja tablicy
const unsigned int tabSize = 5;
int * tab = (int *) malloc(tabSize * sizeof(*tab));
// Rownowaznie tab != NULL, poniewaz NULL == 0
if (tab) {
    for (unsigned int i = 0; i < tabSize; ++i)
        printf("tab[%d]=%d\n", i, tab[i] = i * i);
}

const unsigned int newSize = 10;
int * expandedTab = (int *) realloc(tab, newSize *
   sizeof(*expandedTab));
if (expandedTab) {
    // Wypisz tabSize pierwszych elementow tablicy
    for (unsigned int i = 0; i < tabSize; ++i)
        printf("expandedTab[%d]=%d\n", i,
   expandedTab[i]);
    // Tablica tab zostala zwolniona przez wywolanie
    realloc()
    free(expandedTab);
} else {
    // realloc() zwrocil NULL - tablica tab nie
   zostala zwolniona
    free(tab);
}
```

Listing 8. Realokacja pamięci w języku *C*

Funkcja ***free()*** zwalnia pamięć zaalokowaną za pomocą funkcji ***malloc()***, ***calloc()*** albo ***realloc()***, wskazywaną przez wskaźnik ***ptr***. Jeżeli ***ptr*** jest wskaźnikiem na obszar pamięci zaalokowany przez inną funkcję, to wynikiem działania funkcji ***free()*** jest ***niezdefiniowane zachowanie***. Jeżeli ***ptr*** jest zainicjalizowany jako ***NULL*** (lub ***nullptr***) funkcja nie robi nic. **Funkcja *free()* nie zmienia wartości wskaźnika (w szczególności: nie ustawia wartości na *NULL*) po zwolnieniu wskazywanej przez niego pamięci!!!**

2.3.2. Język *C++*

Ponieważ język *C++* zakłada kompatybilność wsteczną z językiem *C*, to możliwa jest dynamiczna alokacja pamięci z wykorzystaniem funkcji ***malloc()***, ***calloc()***, ***realloc()*** i ***free()***. Jednakże język *C++* oferuje równocześnie nowe (doskonalsze) mechanizmy dynamicznej alokacji pamięci. Stanowią je operatory ***new*** oraz ***delete***. Oba operatory występują w dwóch wersjach, służących do alokacji pamięci dla pojedynczego obiektu oraz dla tablicy. Nagłówki operatorów, zdeklarowane w pliku ***new***, przedstawiono na listingu 9.

```
1 // Alokacja pojedynczego obiektu
2 void * operator new(std::size_t count);
3 void operator delete(void * ptr);
4 // Alokacja tablicy
5 void * operator new[](std::size_t count);
6 void operator delete[](void * ptr);
```

Listing 9. Nagłówki operatorów dedykowanych do dynamicznej alokacji/zwalniania pamięci w języku *C++*

Operator new alokuje dynamicznie pamięć na ***stercie*** o rozmiarze ***count*** bajtów. W przypadku powodzenia zwracany jest **wskaźnik na początek zaalokowanego bloku pamięci**. W innym razie **rzucany jest wyjątek albo zwracany jest wskaźnik *nullptr***. Więcej o obsłudze błędów

operatora new można przeczytać w *Dodatku*.

Operator delete zwalnia pamięć wskazywaną przez wskaźnik ***ptr***, która została uprzednio zaalokowana za pomocą ***operatora new***. Jeżeli ***ptr*** ma wartość ***nullptr***, to ***operator delete*** nie robi nic. **W każdym innym wypadku działanie operatora jest niezdefiniowane**. Z tego względu **niedozwolone jest zwalnianie pamięci zaalokowanej za pomocą funkcji *malloc()*, *calloc()* lub *realloc()* przy użyciu operatora *delete*, jak również zwalnianie pamięci zaalokowanej za pomocą operatora *new* przy użyciu funkcji *free()*. Należy również pamiętać, że operator *delete* nie zmienia wartości wskaźnika (w szczególności: nie ustawia wartości na *nullptr*) po zwolnieniu wskazywanej przez niego pamięci!** Zarówno operator ***new***, jak i operator ***delete*** **mogą być przeciążane**. Przykład dynamicznej alokacji pamięci w języku *C++* przedstawiono na listingu 10. Warto zwrócić szczególną uwagę na postać ***operatorów new i delete*** dla dynamicznej alokacji tablicy.

```
1  // Dynamiczna alokacja pojedynczej zmiennej typu
       calkowitego o wartosci 5
2  int * ptr = new int(5);
3  // Dereferencja wskaznika
4  std::cout << "*ptr=" << *ptr << std::endl;
5  // Zwolnienie pamieci
6  delete ptr;
7
8  // Dynamiczna alokacja tablicy liczb
       zmiennoprzecinkowych
9  const unsigned int tabSize = 5;
10 float * tab = new float[tabSize];
11 // Inicjalizacja i wyswietlenie elementow
12 for (unsigned int i = 0; i < tabSize; ++i)
```

```
13      std::cout << "tab[" << i << "]=" << (tab[i] = i)
        << std::endl;
14  // Zwolnienie pamieci
15  delete[] tab;
```

Listing 10. Dynamiczna alokacja pamięci w języku *C++*

Zalety stosowania ***operatora new*** zamiast funkcji ***malloc()***:

- ***operator new***, w przeciwieństwie do funkcji ***malloc()***, może inicjalizować alokowane zmienne;
- ***operator new***, w przeciwieństwie do funkcji ***malloc()***, może być przeciążany;
- w przypadku funkcji ***malloc()*** wymagane jest jawne podanie rozmiaru alokowanego bloku pamięci (np. przez użycie operatora ***sizeof***).

3. Program ćwiczenia

Zadanie 1. Celem zadania jest określenie wzajemnego położenia segmentów ***stosu*** i ***sterty*** w przestrzeni adresowej komputera. W tym celu utwórz po 4 zmienne alokowane na stosie (statycznie) oraz na stercie (dynamicznie). Następnie zaobserwuj jak zmieniają się adresy kolejnych zmiennych alokowanych w poszczególnych segmentach pamięci. Na tej podstawie **określ wzajemne ułożenie stosu i sterty** (który segment leży wyżej w przestrzeni adresowej) oraz odpowiedz na pytanie: **czy teoretycznie możliwe jest nałożenie się obu segmentów?** (zaobserwuj w którą stronę rozszerza się stos i sterta).

Zadanie 2. Korzystając z języka *C* lub *C++* napisz program, który utworzy macierz kwadratową o wymiarze ***n x n*** elementów i wypełni ją określonymi liczbami. W tym celu pobierz od użytkownika wymiar macierzy ***n***, a następnie n^2 liczb całkowitych, którymi zainicjalizujesz kolejne wyrazy macierzy. Macierz stanowi dynamicznie zaalokowana dwuwymiarowa tablica zmiennych typu ***int***. Po utworzeniu macierzy wypisz na ekranie jej elementy.

Zadanie 3. W pliku nagłówkowym ***arrayUtils.h*** zamieszczono deklarację funkcji ***char * resizeArray(char * array, unsigned int newSize)***. Korzystając z języka *C++* zaimplementuj podaną funkcję, a jej definicję umieść w pliku ***arrayUtils.cpp***. Zadaniem funkcji jest realokacja tablicy znaków ***array***. Funkcja zwraca wskaźnik do nowo zaalokowanej tablicy o rozmiarze ***newSize***. Dodatkowe założenia:

- jeżeli ***array*** ma wartość ***nullptr***, to funkcja ***resizeArray()*** zwraca ***nullptr***;
- funkcja zwalnia pamięć zaalokowaną na tablicę ***array***;

- funkcja kopiuje wartości elementów tablicy ***array*** do nowej tablicy (jeżeli ***newSize*** jest mniejszy niż rozmiar ***array***, to kopiowane jest tylko ***newSize*** pierwszych elementów tablicy ***array***)

Następnie, w funkcji *main()* zaalokuj na stercie tablicę znaków ASCII o początkowym rozmiarze 5 elementów. Wczytuj z klawiatury kolejne znaki ASCII i umieszczaj je w tablicy, aż do dwukrotnego wystąpienia pod rząd tego samego znaku. Jeżeli tablica miałaby się przepełnić, użyj funkcji ***resizeArray()*** i zwiększ jej rozmiar o kolejne 5 elementów. Po zakończeniu wypełniania tablicy znakami, wyświetl utworzony w ten sposób łańcuch na ekranie.

4. Dodatek

4.1. Obsługa błędów operatora *new*

Domyślnie, w przypadku niepowodzenia, ***operator new rzuca wyjątek std::bad_alloc***. ***Wyjątek*** to obiekt służący sygnalizacji, że zaszła nieoczekiwana sytuacja, program napotkał nieprawidłową instrukcję, wystąpił niespodziewany błąd, itp. W języku *C* informacje takie przekazuje się przez zwrócenie z funkcji odpowiedniego kodu błędu (patrz: funkcja *main()*) czy ustawienie odpowiednich ***flag*** (globalnych lub przekazanych do funkcji). Rozwiązania te mają swoje wady, dlatego w języku *C++* wprowadzono ***mechanizm obsługi wyjątków***. Zgłoszenie (*rzucenie*) wyjątku odbywa się zawsze za pośrednictwem wyrażenia ***throw***. W języku *C++* ***wyjątek*** **może być zmienną dowolnego typu**, od typu ***int*** po specjalnie dedykowane ***klasy wyjątków***, których przykładem jest ***std::bad_alloc***:

```
1 throw std::bad_alloc{};
```

Wyjątki obsługiwane są za pomocą dedykowanego bloku ***try–catch***. Funkcję **podejrzewaną o możliwość rzucenia *wyjątku*** umieszcza się w bloku ***try***. W bloku ***catch*** zawieramy instrukcje, które mają zostać wykonane w przypadku **przechwycenia konkretnego *wyjątku*** zgłoszonego przez jedną z funkcji znajdujących się wewnątrz bloku ***try***. Jeżeli ***wyjątek*** nie zostanie rzucony, to program nie wykonuje kodu zawartego wewnątrz bloku ***catch***:

```
1 try {
2     int * tab = new int[1000000000];
3 } catch (const std::bad_alloc &) {
4     std::cout << "Failed to allocate memory" << std
    ::endl;
5 }
```

Zgłoszenie *wyjątku* natychmiastowo przerywa wykonywanie danej funkcji (lub operatora), a sterowanie zwracane jest do funkcji nadrzędnej (wywołującej funkcję zgłaszającą wyjątek). Zatem, rzucony ***wyjątek std::bad_alloc*** przerywa działanie ***operatora new*** i ***propaguje*** do funkcji, która próbowała za jego pomocą zaalokować pamięć. Jeżeli ***wyjątek*** nie zostanie obsłużony (za pomocą bloku ***try-catch***), to propaguje dalej do kolejnych funkcji nadrzędnych, aż do funkcji *main()*. Jeżeli nie zostanie również obsłużony wewnątrz funkcji *main()*, to zostanie wywołana funkcja ***std::terminate()***, która domyślnie wywołuje funkcję ***std::abort()***. Rezultatem jest przerwanie działania programu w wyniku błędu aplikacji, co wiąże się z nieprzeprowadzeniem ***zwijania stosu*** (*ang. stack unwinding*), czego konsekwencją są ***wycieki zasobów*** (w szczególności ***wycieki pamięci***). [*Uwaga: począwszy od standardu C++11 zakłada się, że operator delete (w przeciwieństwie do operatora new) nie rzuca żadnych wyjątków.*]

Możliwe jest jednak wymuszenie na ***operatorze new***, żeby zamiast ***rzucania wyjątku*** w przypadku niepowodzenia, zwrócił wartość ***nullptr***. Wówczas, obsługa błędów alokacji pamięci odbywa się w sposób analogiczny, jak w języku *C*. Aby to zrobić, należy przekazać do (przeciążonego) ***operatora new*** stałą ***std::nothrow***:

```
1 int * tab = new(std::nothrow) int[1000000000];
2 if (tab == nullptr)
3     std::cout << "Failed to allocate memory" << std
  ::endl;
```

4.2. Kilka słów o inteligentnych wskaźnikach

Pisanie zaawansowanych aplikacji w językach *C/C++* nierozłącznie wiąże się z zastosowaniem wskaźników. Jednakże, szczególnie dla początkujących programistów, ich użycie bywa nieintuicyjne i może prowadzić do niejasności. Często skutkuje to wprowadzaniem do kodu niewidocznych na pierwszy rzut oka błędów, których rezultatem są ***wycieki pamięci*** czy ***niezdefiniowane zachowanie***. Podstawowym problemem są niejednoznaczne deklaracje funkcji zwracających wskaźniki. Przykład przedstawiono na listingu 11. Niech ***Config*** stanowi typ zmiennej przechowującej konfigurację aplikacji. Funkcja ***getConfig()*** zwraca wskaźnik na typ ***Config***, który przetwarzany jest przez funkcję ***processConfig()***.

```
1 // Pobranie konfiguracji programu
2 Config * config = getConfig();
3 // Przetworzenie konfiguracji
4 processConfig(config);
5 // Zwolnienie pamieci: delete czy delete[]?
6 delete config;
```

Listing 11. Niejednoznaczności wskaźników – zwracanie wskaźników z funkcji

Problem powstaje w momencie zwalniania pamięci wskazywanej przez ***config***. Patrząc na nagłówek funkcji ***getConfig()*** nie jesteśmy w stanie jednoznacznie określić czy funkcja alokuje pamięć o rozmiarze pojedynczego obiektu (należy zastosować ***operator delete***) czy tablicy obiektów (należy zastosować ***operator delete[]***):

```
1 Config * getConfig();
```

Aby to określić konieczne jest przejrzenie implementacji funkcji ***getConfig()*** (która nie musi być widoczna dla użytkownika – patrz: *biblioteki pro-*

gramistyczne, Ćw. 13):

```
Config * getConfig() {
    return new Config{};
}
```

Dopiero teraz można mieć pewność, że funkcja dynamicznie alokuje pamięć o rozmiarze pojedynczej zmiennej typu ***Config***, zatem zwolnienie pamięci na listigu 11 zostało przeprowadzone poprawnie. Co jednak, gdyby definicja funkcji wyglądała następująco:

```
Config * getConfig() {
    static Config config;
    return &config;
}
```

Funkcja ***getConfig()*** zwracałaby wówczas adres zmiennej statycznej, **bez przeprowadzania dynamicznej alokacji pamięci na stercie.** Zatem zwolnienie pamięci za pomocą ***operatora delete*** byłoby błędem. Co więcej, wywołanie ***operatora delete*** na wskaźniku do zmiennej statycznej kończy się ***niezdefiniowanym zachowaniem***.

Drugi problem z obsługą wskaźników również może wystąpić w kodzie zaprezentowanym na listingu 11. Pozyskany z funkcji ***getConfig()*** wskaźnik przekazywany jest do funkcji ***processConfig()***, w której jest przetwarzany. Następnie zostaje zwolniona pamięć przy użyciu ***operatora delete***. Program nie sprawia wrażenia błędnego – wiadome już jest, że funkcja ***getConfig()*** alokuje pamięć o rozmiarze pojedynczej zmiennej typu ***Config***, zwrócony przez nią wskaźnik jest obsługiwany, a następnie zwalniany przy użyciu poprawnej wersji ***operatora delete***. Co jednak w przypadku, gdy funkcja ***processConfig()*** **rzuci wyjątek**? Działanie funkcji ***processConfig()*** zostanie przerwane, a sterowanie zostanie zwrócone do funkcji wywołującej. Ponieważ w kodzie z listingu 11. wyjątek nie jest obsłużony, to będzie **propagował do**

funkcji nadrzędnej, skutkując niewywołaniem ***operatora delete***. Aby temu zapobiec należałoby poprawić kod tak, jak przedstawiono na listingu 12.

```
Config * config = getConfig();
try {
    // Moze zglosic wyjatek std::runtime_error
    processConfig(config);
    delete config;
} catch (const std::runtime_error &) {
    std::cout << "processConfig() threw an exception
   " << std::endl;
    delete config;
}
```

Listing 12. Niejednoznaczności wskaźników – zwolnienie pamięci przy obsłudze wyjątku

Kolejnym problemem, z jakim można się spotkać, jest weryfikacja wartości przetwarzanego wskaźnika. Przykład przedstawiono na listingu 13. Funkcja ***printConfig()*** przyjmuje wskaźnik na typ ***Config***, sprawdza czy jego wartość jest różna od ***nullptr*** i w przypadku pozytywnej weryfikacji wypisuje na ekranie konfigurację aplikacji.

```
void printConfig(Config * config) {
    if (config != nullptr) {
        // Dereferencja wskaznika config
        std::cout << *config << std::endl;
    }
}
```

Listing 13. Niejednoznaczności wskaźników – weryfikacja wartości wskaźnika

Na pierwszy rzut oka nie widać żadnego błędu w implementacji funkcji. Problem pojawia się w momencie nieopatrznego przekazania do funkcji ***printConfig()*** wskaźnika na obszar pamięci zwolniony przez ***operator delete***:

```
1 Config * config = getConfig();
2 processConfig(config);
3 delete config;
4 // config != nullptr;
5 printConfig(config);
```

Operator delete* nie ustawia zwolnionego wskaźnika na wartość *nullptr*.** W związku z tym, wskaźnik ***config zostanie zweryfikowany przez funkcję ***printConfig()*** jako poprawny (różny od ***nullptr***) i wykonana zostanie na nim dereferencja. **Odwołanie (a więc i dereferencja) do zwolnionego obszaru pamięci skutkuje *niezdefiniowanym zachowaniem*.**

Wskaźnik jest uchwytem (*ang. handle*) do zaalokowanej dynamicznie pamięci. Zadaniem programisty jest zwolnienie pamięci zaalokowanej na stercie. Jeżeli program wyjdzie poza zakres bloku, zdeklarowany w nim wskaźnik zostanie usunięty (zmienna lokalna – automatycznie alokowana na stosie). Pamięć zaalokowana dynamicznie nie będzie mogła być zwolniona (brak uchwytu) – nastąpi ***wyciek pamięci***. Rozwiązaniem jest opakowanie „surowego" wskaźnika (*ang. raw pointer*) w obiekt automatycznie zwalniający pamięć podczas destrukcji. Oznacza to **przekazanie zarządzania pamięcią alokowaną na stercie do obiektu alokowanego na stosie**. Taki mechanizm został wprowadzony od standardu *C++*11 pod nazwą ***inteligentne wskaźniki*** (*ang. smart pointers*) i stanowi jeden z ważniejszych elementów ***nowoczesnego programowania obiektowego*** w języku *C++*. ***Inteligentne wskaźniki*** stanowią trzy klasy: ***std::unique_ptr***, ***std::shared_ptr*** oraz ***std::weak_ptr***. Przyjmują one w ***konstruktorach*** (specjalnych metodach wywoływanych w momencie tworzenia obiektu) ***wskaźniki*** na pamięć zaalokowaną dynamicznie. W momencie usuwania ***inteligentnego wskaźnika***

wywoływana jest ***metoda*** zwana ***destruktorem***. Zadaniem ***destruktora*** (w tym przypadku) jest zwolnienie dynamicznie zaalokowanej pamięci. Stanowi to realizację ***wzorca projektowego RAII – pozyskanie zasobu jest inicjalizacją*** (*ang. Resource Acquisition Is Initialization*). Operowanie na właściwie zainicjalizowanym ***inteligentnym wskaźniku*** gwarantuje pracę na poprawnym *surowym wskaźniku*. Programista nie musi jawnie zwalniać dynamicznie zaalokowanej pamięci – odpowiedzialność za to przejmuje obiekt ***inteligentnego wskaźnika***. Co więcej, pamięć zostanie zwolniona również, jeżeli w kodzie przetwarzającym ***inteligentny wskaźnik*** zostanie ***zgłoszony wyjątek***, ponieważ przy usuwaniu obiektu ***inteligentnego wskaźnika*** wywołany zostanie ***destruktor***. Warunkiem jest poprawne ***zwinięcie stosu***, do czego wymagane jest **przechwycenie *wyjątku*** (najpóźniej w funkcji *main()*).

4.3. Dynamiczna analiza programu pod kątem wycieków pamięci

Komercyjne projekty programistyczne wykorzystują wiele narzędzi umożliwiających ***dynamiczną analizę programów*** (analizę w trakcie ich działania). Przykładem takich narzędzi są np. ***debuggery***. W kontekście ***wskaźników*** i ***dynamicznej alokacji pamięci*** szczególną rolę pełnią programy umożliwiające ***dynamiczną analizę kodu pod kątem wycieków pamięci***. Jednym z najpopularniejszych analizatorów tego typu (dedykowany do pracy na systemach *Unix*) jest ***Valgrind***. Na listingu 14. przedstawiono prosty program (plik *main.cpp*) prezentujący wyciek pamięci.

```
1 #include <iostream>
2
3 int main() {
4     const unsigned int size = 5;
5     int * tab = new int[size];
```

```
6       for (unsigned int i = 0; i < size; ++i) {
7       std::cout << "tab[" << i << "]=" << (tab[i] = i)
        << std::endl;
8       }
9       // Wyciek pamieci - brak delete[] tab
10      return 0;
11  }
```

Listing 14. Wyciek pamięci

Aby poddać program analizie ***Valgrinda*** należy go najpierw skompilować, rezygnując z optymalizacji, np. tak: ***g++ main.cpp -o app -O0***. Uruchamiając analizę (***valgrind -v --leak-check=full --show-leak-kinds=all ./app***) otrzymujemy następujący rezultat:

```
1  ==28856== HEAP SUMMARY:
2  ==28856==     in use at exit: 72,724 bytes in 2
       blocks
3  ==28856==   total heap usage: 3 allocs, 1 frees,
       73,748 bytes allocated
4  ==28856==
5  ==28856== Searching for pointers to 2 not-freed
       blocks
6  ==28856== Checked 107,520 bytes
7  ==28856==
8  ==28856== 20 bytes in 1 blocks are definitely lost
       in loss record 1 of 2
9  ==28856==    at 0x4C2E80F: operator new[](unsigned
       long) (in /usr/lib/valgrind/vgpreload_memcheck-
       amd64-linux.so)
10 ==28856==    by 0x4008EA: main (in /home/michau/
       workspace/test/app)
```

```
==28856==
==28856== 72,704 bytes in 1 blocks are still
   reachable in loss record 2 of 2
==28856==    at 0x4C2DB8F: malloc (in /usr/lib/
   valgrind/vgpreload_memcheck-amd64-linux.so)
==28856==    by 0x4EDBF85: ??? (in /usr/lib/x86_64-
   linux-gnu/libstdc++.so.6.0.28)
==28856==    by 0x40106C9: call_init.part.0 (dl-init
   .c:72)
==28856==    by 0x40107DA: call_init (dl-init.c:30)
==28856==    by 0x40107DA: _dl_init (dl-init.c:120)
==28856==    by 0x4000C69: ??? (in /lib/x86_64-linux
   -gnu/ld-2.23.so)
==28856==
==28856== LEAK SUMMARY:
==28856==    definitely lost: 20 bytes in 1 blocks
==28856==    indirectly lost: 0 bytes in 0 blocks
==28856==      possibly lost: 0 bytes in 0 blocks
==28856==    still reachable: 72,704 bytes in 1
   blocks
==28856==         suppressed: 0 bytes in 0 blocks
==28856==
==28856== ERROR SUMMARY: 1 errors from 1 contexts (
   suppressed: 0 from 0)
==28856== ERROR SUMMARY: 1 errors from 1 contexts (
   suppressed: 0 from 0)
```

Jak widać, ***Valgrind*** wskazuje na wyciek pamięci (*20 bytes in 1 blocks are definitely lost in loss record 1 of 2 at 0x4C2E80F: operator new[](unsigned long)*). Po uwzględnieniu wywołania ***operatora delete[]*** w kodzie z listingu 14. i ponownej kompilacji programu otrzymamy następujący wynik analizy:

```
==29405== HEAP SUMMARY:
==29405==     in use at exit: 72,704 bytes in 1 blocks
==29405==   total heap usage: 3 allocs, 2 frees, 73,748 bytes allocated
==29405==
==29405== Searching for pointers to 1 not-freed blocks
==29405== Checked 107,520 bytes
==29405==
==29405== 72,704 bytes in 1 blocks are still reachable in loss record 1 of 1
==29405==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==29405==    by 0x4EDBF85: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28)
==29405==    by 0x40106C9: call_init.part.0 (dl-init.c:72)
==29405==    by 0x40107DA: call_init (dl-init.c:30)
==29405==    by 0x40107DA: _dl_init (dl-init.c:120)
==29405==    by 0x4000C69: ??? (in /lib/x86_64-linux-gnu/ld-2.23.so)
==29405==
==29405== LEAK SUMMARY:
==29405==    definitely lost: 0 bytes in 0 blocks
==29405==    indirectly lost: 0 bytes in 0 blocks
==29405==      possibly lost: 0 bytes in 0 blocks
==29405==    still reachable: 72,704 bytes in 1 blocks
==29405==         suppressed: 0 bytes in 0 blocks
==29405==
```

```
==29405== ERROR SUMMARY: 0 errors from 0 contexts (
   suppressed: 0 from 0)
==29405== ERROR SUMMARY: 0 errors from 0 contexts (
   suppressed: 0 from 0)
```

Literatura

[1] S. Prata. *Język C++. Szkoła programowania*. 6th ed. Helion, 2013. ISBN: 978-83-246-4336-3.