



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

<b>1</b>	<b>Cel ćwiczenia</b>	<b>2</b>
<b>2</b>	<b>Wprowadzenie</b>	<b>2</b>
2.1	Referencje . . . . .	2
2.2	Stos i sverta . . . . .	10
2.3	Dynamiczna alokacja pamięci . . . . .	12
2.3.1	Język <i>C</i> . . . . .	12
2.3.2	Język <i>C++</i> . . . . .	16
<b>3</b>	<b>Program ćwiczenia</b>	<b>19</b>
<b>4</b>	<b>Dodatek</b>	<b>21</b>
4.1	Obsługa błędów operatora <i>new</i> . . . . .	21
4.2	Kilka słów o inteligentnych wskaźnikach . . . . .	23
4.3	Dynamiczna analiza programu pod kątem wycieków pamięci .	27

---

## 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 zadeklarowanych 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
```

```

9 // Modyfikacja wartosci zmiennej value z
   wykorzystaniem referencji
10 refValue = 5;
11
12 // Wypisze wartosc 5
13 std::cout << value << std::endl;
14 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**:

```

1 int value = 2;
2 int & refValue = value;
3
4 // Wypisze te sama wartosc adresu
5 std::cout << &value << std::endl;
6 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.

```
1 // int & to referencja (y jest referencja x)
2 int & y = x;
3
4 // int * to wskaźnik (ptr jest wskaźnikiem do val)
5 // &val to operator adresu na zmiennej val
6 int * ptr = &val;
7
8 // *ptr to operator dereferencji na wskaźniku ptr
9 std::cout << *ptr << std::endl;
10
11 // *ptr to operator dereferencji na wskaźniku ptr
12 // symbol * nie jest bezpośrednio poprzedzony typem
   // zmiennej - występuje nawias
13 std::cout << (float)*ptr << std::endl;
14
15 // long * to wskaźnik (rzutowanie typów)
16 // identyfikator zmiennej nie jest bezpośrednio
   // poprzedzony symbolem * - występuje nawias
17 long * ptrCast = (long *)ptr;
```

Listing 2. Różne znaczenia symboli & i \*

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

```
1 float val = 3.5;
2 float & ref; // Błąd kompilacji
3 ref = val;
```

---

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

```
1 float val = 3.5;
2 float * const ptr; // Bład kompilacji
3 ptr = &val; // Bład kompilacji
```

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

```
1 long x = 123;
2 long & ref = x;
3
4 long y = 654;
5 // Przypisanie ref (oraz x) wartosci 654
6 ref = y;
```

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

```
1 long x = 123;
2 long * const ptr = &x;
3
4 long y = 654;
5 ptr = &y; // Bład 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 kopii zmiennych (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 oryginału
5     ++valRef;
6 }
7
8 int main() {
9     int val = 111;
10    // inicjalizacja zmiennej valRef i modyfikacja
    zmiennej val
11    incrementVal(val);
12    // Wyświetla 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     // Bład kompilacji - modyfikacja stałej
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     // Bład kompilacji - funkcja mogłaby
17     // zmodyfikować stałą
18     useReference(constVal);
19     return 0;
20 }
```

Listing 4. Stała referencja jako argument funkcji



---

**Stałe referencje** mają jeszcze jeden istotny aspekt. Możliwe jest wywołanie 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
10    // mozliwosci odwołania sie, pobrania adresu
11    std::cout << factorial(3) << std::endl;
```

```
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.<sup>[1]</sup>

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*):

---

```
1 #include <iostream>
2
3 int & getReference() {
4     // Zmienna lokalna
5     int var = 10;
6     return var;
7 }
8
9 int main() {
10     // Dyndająca referencja
11     int & ref = getReference();
12     // Niezdefiniowane zachowanie!
13     std::cout << ref << std::endl;
14     return 0;
15 }
```

## 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 **przepiętnienia 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/operatorsy. **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. Przeciwnieństwem 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 zadeklarowane w nagłówku ***stdlib.h***; ich nagłówki zestawiono na listingu 6. Żadnej z wymienionych funkcji **nie można przeciążyć**. 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.

```
1 void * malloc(size_t size);
2 void * calloc(size_t num, size_t size);
3 void * realloc(void * ptr, size_t new_size);
4 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.]

```
1 const unsigned int tabSize = 5;
2 // Dynamiczna alokacja 5-elementowej tablicy liczb
   całkowitych
3 // Alokowany rozmiar stanowi iloczyn tabSize i
   rozmiaru pojedynczego elementu tablicy tab (int)
4 // Analogicznie: malloc(tabSize * sizeof(int))
5 int * tab = (int *) malloc(tabSize * sizeof(*tab));
6 if (tab != NULL) {
7     // 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.

```
1 // Dynamiczna alokacja tablicy
2 const unsigned int tabSize = 5;
3 int * tab = (int *) malloc(tabSize * sizeof(*tab));
4 // Równowaznie tab != NULL, poniewaz NULL == 0
5 if (tab) {
6     for (unsigned int i = 0; i < tabSize; ++i)
7         printf("tab[%d]=%d\n", i, tab[i] = i * i);
8 }
9
10 const unsigned int newSize = 10;
11 int * expandedTab = (int *) realloc(tab, newSize *
    sizeof(*expandedTab));
12 if (expandedTab) {
13     // Wypisz tabSize pierwszych elementow tablicy
14     for (unsigned int i = 0; i < tabSize; ++i)
15         printf("expandedTab[%d]=%d\n", i,
    expandedTab[i]);
16     // Tablica tab zostala zwolniona przez wywołanie
    realloc()
17     free(expandedTab);
18 } else {
19     // realloc() zwrocil NULL - tablica tab nie
    zostala zwolniona
20     free(tab);
21 }
```

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 **zgłaszany jest wyjątek** albo zwracany jest wskaźnik ***nullptr***. Więcej o obsłudze błędów

---

dów *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
   całkowitego 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 inicjować 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 sterpie (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  $n \times n$  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 stacku 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* zgłasza *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  
5     ::endl;  
6 }
```

---

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 pamięci: 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 przejrzanie implementacji funkcji *getConfig()* (która nie musi być widoczna dla użytkownika – patrz: *biblioteki pro-*



---

gramistyczne, Ćw. 13):

```
1 Config * getConfig() {  
2     return new Config{};  
3 }
```

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

```
1 Config * getConfig() {  
2     static Config config;  
3     return &config;  
4 }
```

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()** zgłosi wyjątek? Działanie funkcji **processConfig()** zostanie przerwane, a sterowanie zostanie zwrócone do funkcji wywołującej. Ponieważ w kodzie z listingu 11. wyjątek nie jest obsługiwany,

---

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

```
1 Config * config = getConfig();
2 try {
3     // Moze zglosic wyjatek std::runtime_error
4     processConfig(config);
5     delete config;
6 } catch (const std::runtime_error &) {
7     std::cout << "processConfig() threw an exception
8     " << std::endl;
9     delete config;
10 }
```

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.

```
1 void printConfig(Config * config) {
2     if (config != nullptr) {
3         // Dereferencja wskaznika config
4         std::cout << *config << std::endl;
5     }
6 }
```

Listing 13. Niejednoznaczności wskaźników – weryfikacja wartości wskaźnika

---

Na pierwszy rzut oka nie widać żadnego błędu w implementacji funkcji. Problem pojawia się w momencie nieopatrzego 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 sterce. Jeżeli program wyjdzie poza zakres bloku, zadeklarowany 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 sterce 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***. ***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 *destruktor*. Zadaniem *destruktor* (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 *zwiniecie 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)

```

```

11 ==28856==
12 ==28856== 72,704 bytes in 1 blocks are still
    reachable in loss record 2 of 2
13 ==28856==      at 0x4C2DB8F: malloc (in /usr/lib/
    valgrind/vgpreload_memcheck-amd64-linux.so)
14 ==28856==      by 0x4EDBF85: ??? (in /usr/lib/x86_64-
    linux-gnu/libstdc++.so.6.0.28)
15 ==28856==      by 0x40106C9: call_init.part.0 (dl-init
    .c:72)
16 ==28856==      by 0x40107DA: call_init (dl-init.c:30)
17 ==28856==      by 0x40107DA: _dl_init (dl-init.c:120)
18 ==28856==      by 0x4000C69: ??? (in /lib/x86_64-linux
    -gnu/ld-2.23.so)
19 ==28856==
20 ==28856== LEAK SUMMARY:
21 ==28856==      definitely lost: 20 bytes in 1 blocks
22 ==28856==      indirectly lost: 0 bytes in 0 blocks
23 ==28856==      possibly lost: 0 bytes in 0 blocks
24 ==28856==      still reachable: 72,704 bytes in 1
    blocks
25 ==28856==      suppressed: 0 bytes in 0 blocks
26 ==28856==
27 ==28856== ERROR SUMMARY: 1 errors from 1 contexts (
    suppressed: 0 from 0)
28 ==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:

---

```
1 ==29405== HEAP SUMMARY:
2 ==29405==      in use at exit: 72,704 bytes in 1
      blocks
3 ==29405==    total heap usage: 3 allocs, 2 frees,
      73,748 bytes allocated
4 ==29405==
5 ==29405== Searching for pointers to 1 not-freed
      blocks
6 ==29405== Checked 107,520 bytes
7 ==29405==
8 ==29405== 72,704 bytes in 1 blocks are still
      reachable in loss record 1 of 1
9 ==29405==    at 0x4C2DB8F: malloc (in /usr/lib/
      valgrind/vgpreload_memcheck-amd64-linux.so)
10 ==29405==    by 0x4EDBF85: ??? (in /usr/lib/x86_64-
      linux-gnu/libstdc++.so.6.0.28)
11 ==29405==    by 0x40106C9: call_init.part.0 (dl-init
      .c:72)
12 ==29405==    by 0x40107DA: call_init (dl-init.c:30)
13 ==29405==    by 0x40107DA: _dl_init (dl-init.c:120)
14 ==29405==    by 0x4000C69: ??? (in /lib/x86_64-linux
      -gnu/ld-2.23.so)
15 ==29405==
16 ==29405== LEAK SUMMARY:
17 ==29405==    definitely lost: 0 bytes in 0 blocks
18 ==29405==    indirectly lost: 0 bytes in 0 blocks
19 ==29405==    possibly lost: 0 bytes in 0 blocks
20 ==29405==    still reachable: 72,704 bytes in 1
      blocks
21 ==29405==          suppressed: 0 bytes in 0 blocks
22 ==29405==
```

---

```
23 ==29405== ERROR SUMMARY: 0 errors from 0 contexts (
    suppressed: 0 from 0)
24 ==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.