Programowanie I
Wykład 6

dr inż. Rafał Brociek

Wydział Matematyki Stosowanej
Politechnika Śląska

15.11.2021

Rodzaje rzutowań w C++

W C++ rozróżniać będziemy następujące rodzaje rzutowań:

```
1 static_cast<nazwa_typu>(wyrażenie);
2
3 const_cast<nazwa_typu>(wyrażenie);
4
5 dynamic_cast<nazwa_typu>(wyrażenie);
6
7 reinterpret_cast<nazwa_typu>(wyrażenie);
```

Rzutowanie static_cast

Do rzutowania zmiennych możemy posłużyć się operatorem:

static_cast<*typ*>(*wyrażenie*) - służy do konwersji „możliwych” (również stratnych), gdzie wyrażenie zostaje zrzutowane na typ ujęty w nawiasach ostrych.

```
1 double zmienna = 9.87;
2
3 // rzutowanie w starym stylu
4 int a = (int)zmienna;
5
6 // rzutowanie według standardu
7 int b = static_cast<int>(zmienna);
8
9 cout << a << " " << b << endl; // 9
```

Rzutowanie const_cast

W przypadku, gdy chcemy stałą (const) zrzutować na zmienną (tego samego typu) korzystamy z operatora rzutowania const_cast. **Uwaga**. Używając operatora const_cast należy operować na wskaźnikach.

```
1  int fun(int* wsk)
2  {
3      return *wsk + 10;
4  }
5  int main()
6  {
7      const int wartosc = 15;
8      const int *wsk = &wartosc;
9
10     //int* wsk2 = wsk; // błąd
11     //int nowaWartosc = const_cast<int>(wartosc); // błąd
12     // rzutowanie
13     int *nowyWsk = const_cast <int *>(wsk);
14     // cout << fun(wsk) << endl; błąd
15     cout << fun(nowyWsk) << endl;
16 }
```

Rzutowanie `reinterpret_cast`

Operator rzutowania `reinterpret_cast` służy do konwersji wskaźników. Umożliwia również rzutowanie ze wskaźnika na typ całkowity (`int`).

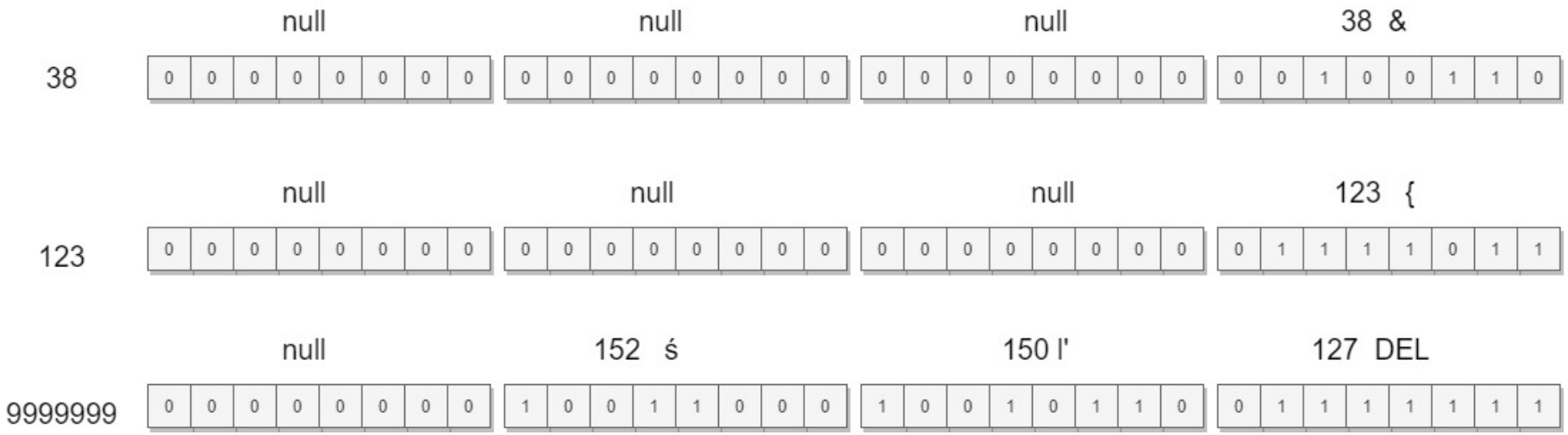
Uwaga. Będzie wykorzystywany przy zapisie/odczycie danych do/z pliku w postaci binarnej.

Rzutowanie reinterpret_cast

```
1  unsigned int tabInt[] = { 38, 123, 9999999 };
2  unsigned int *wsk_UI = tabInt;
3
4  // wypisujemy liczby
5  for (int i = 0; i < 3; i++)
6     cout << *(wsk_UI + i) << endl;
7  // rzutowanie na wskaźnik typu unsigned char*
8  unsigned char *wsk_UCh = nullptr;
9  wsk_UCh = reinterpret_cast<unsigned char*>(wsk_UI);
10
11 for (int i = 0; i < 12; i++)
12 {
13    cout << "bajt " << i << ": " << "\t";
14    cout << "adres w pamieci: " << setw(10) <<
15       reinterpret_cast<int>(wsk_UCh + i) << " ";
16    cout << "kod: " << setw(3) << (int)(*(wsk_UCh + i)) << " ";
17    cout << " znak: " << setw(3) << *(wsk_UCh + i) << endl;
18 }
19
20 // kodowanie cp852 unsigned int - 4 bajty, char - 1 bajt
21 //liczby z tablicy przedstawić w postaci binarnej
```

Rzutowanie `reinterpret_cast`

`https://www.ascii-codes.com/cp852.html`



Rysunek: Rzutowanie wskaźników

Wskaźnik posiada wiedzę o adresie oraz typie danych. Inaczej jest w przypadku wskaźników `void*`.

Wskaźniki typu `void*`

Wskaźnik typu `void*` przekazuje tylko adres, nie zawiera informacji o typie danych, na które wskazuje. Można do niego przypisać dowolny inny wskaźnik (obiektu non-const) bez rzutowania. Można nim pokazywać daną dowolnego typu.

```
1  int tab[10] = {1,1,1,1};
2  int *wsk_int = &tab[9];
3  double d = 10;
4  bool b = false;
5
6  // nie potrzeba rzutowania
7  void *wsk_v = wsk_int;
8  wsk_v = &d;
9  wsk_v = &b;
10 ...
11
12 // w drugą stronę rzutowanie niezbędne
13 wsk_int = reinterpret_cast<int*>( wsk_v );
```

Dynamiczne przydział pamięci

Sterta (heap) – obszar wolnej pamięci:

- przeznaczony na dane dynamiczne,
- kontrolowany ręcznie przez programistę,
- ograniczony.

Stos (stack) – obszar pamięci roboczej:

- przeznaczony na dane automatyczne (zmienne lokalne),
- nie jest bezpośrednio kontrolowany przez programistę,
- jest strukturą danych działających na zasadzie LIFO (last in first out),
- ograniczony.

Dynamiczne przydział pamięci

Dynamiczny przydział (alokacja) pamięci polegać będzie na zarezerwowaniu fragmentu pamięci z obszaru sterty w trakcie działania programu (stąd nazwa dynamiczny).

- Programista ustala wielkość obszaru jaki chce zarezerwować,
- adres początku zarezerwowanego obszaru należy zapisać w zmiennej wskaźnikowej,
- programista musi zadbać o zwolnienie pamięci, gdy nie jest już ona potrzebna.

Dynamiczne lokowanie pamięci w języku C

Funkcje calloc i malloc

```
1 //inicjuje pamięc zerami
2 void* calloc(ilosc obiektow, rozmiar obiektu);
3
4 void* malloc(rozmiar_w_bajtach);
```

```
1 int *wsk_int = (int*)calloc (1000, sizeof(int));
2 ...
3 //zwolnienie pamięci!
4 free(wsk_int);
5 void *wsk_v = malloc(1000000);
6 ...
7 free(wsk_v);
```

Dynamiczne lokowanie pamięci w języku C

```
1 int n = 100000;
2 int *p = (int*)malloc(n * sizeof(int));
3
4 p[10] = 12;
5 cout << (*(p+10)) << endl;
6
7 free(p);
```

Uwaga!!! Pamięć dynamicznie przydzielona nie jest automatycznie zwalniana, nawet jeśli zmienna lokalna `p` przestanie istnieć. Funkcja `free` zwalnia pamięć wskazywaną przez `p`, lub nic nie robi jeśli `p` jest `nullptr`. Można zwalniać jedynie bloki pamięci spod adresów uzyskanych funkcjami `malloc` lub `calloc`.

Dynamiczne lokowanie pamięci w języku C

```
1  #include <thread> // do sleep_for
2
3  void funkcja(void)
4  {
5     int *wsk = (int*)malloc( 8 * sizeof( int ) );
6  }
7  int main()
8  {
9     for(int i = 0 ; i < 1000 ; i++ )
10       funkcja();
11       this_thread::sleep_for(1s);
12 }
13
14 // W VS Diagnostyka -> Użycie pamięci (Profilowanie sterty -> Utw↩
       órz migawkę)
```

Jeśli „zgubimy” adres dynamicznie przydzielonej pamięci, to już jej nie zwolnimy. Dochodzi to tzw. wycieku pamięci.

Dynamiczne lokowanie pamięci w języku C++

W języku C++ do dynamicznej rezerwacji oraz zwalnianiu pamięci będziemy korzystać z operatorów `new` oraz `delete`.

Funkcje new i delete – schemat

```
1 // rezerwowanie pamięci
2 wskaznik = new typZmiennej;
3 wskaznik = new typZmiennej[iloscElementow];
4
5 // zwalnianie pamięci
6 delete wskaznik;
7 delete[] wskaznikDoTablicy;
```

Uwaga. Operator `new` generuje wyjątek w przypadku braku wolnej pamięci. Więcej o obsłudze sytuacji wyjątkowych (`try, catch`) na późniejszym wykładzie.

Dynamiczne lokowanie pamięci w języku C++

```
1  int main()
2  {
3      int n{};
4      cout << "Ile liczb potrzebujesz?" << endl;
5      cin >> n;
6
7      // dynamiczna rezerwacja pamięci
8      int* tab = nullptr;
9      tab =  new int[n];
10
11     for (int i = 0; i < n; i++)
12     {
13         cout << endl << "tab [" << i << "] = ";
14         cin >> tab[i];
15     }
16     cout << tab[n - 2] << endl;
17
18     // zwolnienie zarezerwowanej wcześniej pamięci
19     delete[] tab;
20     tab = nullptr; // dobra praktyka
21 }
```

Wskaźnik do funkcji

Wskaźnik do funkcji - deklaracja i definicja

```
1  int funkcja();
2  double funkcja2(int, int);
3  void funkcja3(char*, int, bool);
4  ...
5  // wskaźnik do funkcji o prototypie int f()
6  int(*wFun1)() = nullptr;
7  wFun1 = &funkcja;
8  // wskaźnik do funkcji
9  // o prototypie double f(int, int)
10 double(*wskFun2)(int, int) {};
11 wskFun2 = &funkcja2;
12 // wskaźnik do funkcji
13 // o prototypie void f(char*, int, bool)
14 void(*wskFun3)(char*, int, bool) {};
15 wskFun3 = funkcja3;
```

Wskaźnik do funkcji

```
1  int suma(int, int);
2  int iloczyn(int, int);
3
4  int main()
5  {
6      int(*dzialanie)(int, int) {}; // int(*dzialanie)(int, int) = ↩
           nullptr;
7      bool czyDodawac = false;
8      int a = 10, b = 21;
9      if (czyDodawac)
10         dzialanie = &suma;
11     else
12         dzialanie = &iloczyn;
13     cout << dzialanie(a, b) << endl;
14 }
15
16 int suma(int a, int b) { return a + b; }
17 int iloczyn(int a, int b) { return a * b; }
```

Wskaźnik do funkcji jako argument funkcji

Różne normy wektorów ($\mathbf{x} = (x_1, x_2, \ldots, x_n)$):

1. norma $L1$: $||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i|$,
2. norma $L2$: $||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$,
3. norma maksimum: $||\mathbf{x}||_\infty = \max\{|x_1|, |x_2|, \ldots, |x_n|\}$,

Zadanie. Napisać funkcję:
`void normaW(double( *norma )(double*, int), double* w, int dl)`
realizującą obliczanie normy wektora, gdzie pierwszy argument jest wskaźnikiem do funkcji.

Wskaźnik do funkcji jako argument funkcji

```
1  double normaL2(double* wektor, int dl);
2  double normaL1(double* wektor, int dl);
3  double normaMax(double* wektor, int dl);
4  double normaW(double(*norma)(double*, int), double* w, int dl);
5  int main()
6  {
7      double* wektor{};
8      wektor = new double[3];
9      wektor[0] = 5;
10     wektor[1] = 1;
11     wektor[2] = 2;
12     cout << endl << "L1 = " << normaW(normaL1, wektor, 3) << endl;
13     cout << endl << "L2 = " << normaW(normaL2, wektor, 3) << endl;
14     cout << endl << "max = " << normaW(normaMax, wektor, 3) << ↩
           endl;
15     delete [] wektor;
16 }
17
18 double normaW(double(*norma)(double*, int), double* w, int dl)
19 {
20     return norma(w, dl);
21 }
```

Podsumowanie cz. I

- W C++ rozróżniamy cztery typy rzutowań:
  - `static_cast`,
  - `const_cast`,
  - `reinterpret_cast`,
  - `dynamic_cast`
- `Wskaźnik typu void* zawiera informację tylko o adresie, o typie już nie. Wskaźnikiem tego typu możemy pokazywać dane dowolnego typu.`
- Sterta (heap) jest przeznaczona na dane dynamiczne (rezerwowane w sposób dynamiczny), zaś stos (stack) przeznaczony jest na dane automatyczne (zmienne lokalne).
- Przy dynamicznej rezerwacji pamięci w C++ programista ustala wielkość obszaru do rezerwacji, musi również pamiętać o zwolnieniu tego obszaru.

Podsumowanie cz. II

- W języku C pamięć w sposób dynamiczny rezerwuje się przy użyciu funkcji `calloc`, `malloc`, zaś zwalnia przy użyciu funkcji `free`.
- W języku C++ pamięć w sposób dynamiczny rezerwuje się przy użyciu operatora `new`, zaś zwalnia przy użyciu operatora `delete`. Operator `new` może wyrzucić wyjątek.
- Uważać na tzw. „wyciek pamięci”.
- `double (*wskDoFun)(int a, bool b) = nullptr;` – wskaźnik do funkcji zwracającej wartość typu `double` i przyjmującej jako argumentu wartości typu `int` oraz `bool`.

Dziękuję za uwagę