Wykład nr 4

Temat: Dynamiczny przydział pamięci, zastosowania wskaźników, praca z plikami.

Cytaty:

Zamiast dzielić ludzi na kobiety i mężczyzn,
powinniśmy dzielić ich na statycznych i dynamicznych.

Charles Churchill

Adresy zostały nam dane po to,
aby ukryć miejsce naszego pobytu.

Saki (H.H. Munro)

Mogę tylko przypuszczać, że dokument „Nie archiwizować” jest zarchiwizowany w archiwum „Nie archiwizować”.

Senator Frank Church

Wykład nr 4

Temat: Dynamiczny przydział pamięci, zastosowania wskaźników, praca z plikami.

Zakres wykładu:

- dynamiczny przydział pamięci, operatory **`new`** i **`delete`**
- tablice dynamiczne
- tablice wskaźników
- napisy, tablice napisów
- wskaźniki do funkcji
- tablice wskaźników do funkcji
- operacje wejścia/wyjścia
- formatowanie
- operacje na plikach dyskowych
- podsumowanie
- ćwiczenia powtórzeniowe i sprawdzające
- następny wykład

Dynamiczny przydział pamięci, operatory **`new`** i **`delete`**

Operatory **new** i **delete** służą do dynamicznego alokowania pamięci operacyjnej komputera. Mogą być wykorzystane dla dowolnego typu danych, tj. wbudowanego lub zdefiniowanego przez użytkownika.

Dynamiczne przydzielanie pamięci umożliwia programiście tworzenie i usuwanie zmiennych, a tym samym pełne sterowanie czasem ich istnienia. Zmienne przechowywane są w specjalnie wydzielonym obszarze pamięci do swobodnego używania (ang. *heap* – zapas, ang. *free store* – swobodnie dostępny magazyn).

Do przydzielania i zwalniania pamięci służą odpowiednio słowa kluczowe **new** i **delete**. Za pomocą operatora **delete** kasuje się tylko obiekty stworzone operatorem **new**. Dostęp do zmiennych dynamicznych umożliwiają wskaźniki.

Cechy obiektów stworzonych operatorem new:

1. obiekty istnieją od momentu ich utworzenia operatorem **new** do momentu skasowania operatorem **delete**, **to my decydujemy o czasie życia obiektów**
2. **obiekty takie nie mają nazwy**, można nimi operować tylko za pomocą wskaźników
3. **obiektów tych nie obowiązują zwykłe zasady o zakresie ważności**, czyli to, w których miejscach programu są widzialne lub niewidzialne
4. w obiektach dynamicznych (tworzonych operatorem **new**) zaraz po ich utworzeniu tkwią jeszcze „śmieci” (przypadkowe wartości), musimy zatem **zadbać o wstępną inicjalizację tych obiektów**

Przykład 1:

```
float *nazwa;              //definicja wskaźnika do typu float
nazwa = new float;         //utworzenie obiektu float o odpowiedniej wielkości

LUB
float *nazwa = new float; //można w jednej linii zaalokować pamięć

*nazwa=1;                  //zapisanie wartości 1 do obiektu typu float

delete nazwa;              //zwolnienie pamięci zajmowanej przez obiekt
nazwa=0;                   //odłączenie wskaźnika od pamięci
```

Przykład 2:

```
int *wsk;                  //definicja wskaźnika do typu int
wsk = new int(10);         //inicjalizacja dynamicznie zaalokowanej pamięci

LUB
int *wsk = new int(10);    //można w jednej linii zaalokować pamięć

cout<<*wsk<<endl;          //wyświetlenie zawartości obiektu

delete wsk;                //zwolnienie pamięci zajmowanej przez obiekt
wsk=0;                     //odłączenie wskaźnika od pamięci
```

UWAGA: Po zwolnieniu zaalokowanej wcześniej pamięci, nadaj wskaźnikowi, którym się posługiwałeś wartość 0, aby mieć do niej dostęp. Spowoduje to odłączenie wskaźnika od tego obszaru pamięci.

Tablice dynamiczne

TABLICE JEDNOWYMIAROWE

Przykład 1:

```
float *t;                   //definicja wskaźnika do typu float
t = new float[10];          //utworzenie 10-elementowej tablicy typu float
```

LUB

```
float *t = new float[10]; //można w jednej linii zaalokować tablicę

for(int i=0;i<10;i++)
   cin>>t[i];               //wczytanie danych do tablicy

delete [] t;                //zwolnienie pamięci zajmowanej przez tablicę
t=0;                        //odłączenie wskaźnika od pamięci
```

Przykład 2:

```
int *tab,rozmiar=5;         //definicja wskaźnika do typu int oraz zmiennej int
tab = new int[rozmiar];     //dynamiczne zaalokowanie pamięci o podanym rozmiarze
```

LUB

```
int *tab = new int[rozmiar];       //można w jednej linii zaalokować pamięć

for(int i=0;i<rozmiar;i++)
   cout<<tab[i]<<endl;     //wyświetlenie zawartości tablicy

delete [] tab;              //zwolnienie pamięci zajmowanej przez tablicę
tab=0;                      //odłączenie wskaźnika od pamięci
```

UWAGA: Po zwolnieniu zaalokowanej wcześniej pamięci, nadaj wskaźnikowi, którym się posługiwałeś wartość 0, aby mieć do niej dostęp. Spowoduje to odłączenie wskaźnika od tego obszaru pamięci.

TABLICE DWUWYMIAROWE

Przykład:

```
int m = 3;                                  //liczba wierszy
int n = 5;                                  //liczba kolumn
int **macierz;


macierz = new int*[m];                      //KROK 1: alokacja wierszy

for (int j=0;j<m;j++)
   macierz[j]=new int[n];                   //KROK 2: alokacja kolumn


for(int i=0;i<m;i++)
   for(int j=0;j<n;j++)
      cin>>macierz[i][j];                   //wczytanie danych do macierzy


for (int i=0;i<m;i++)
   delete[] macierz[i];                     //KROK 1: usuwanie kolumn

delete[] macierz;                           //KROK 2: usuwanie wierszy


macierz=NULL;                               //odłączenie wskaźnika od pamięci
```

Tablice wskaźników

Tablice mogą zawierać wskaźniki (czyli adresy). Są to tzw. **tablice wskaźników**.

Przykład:

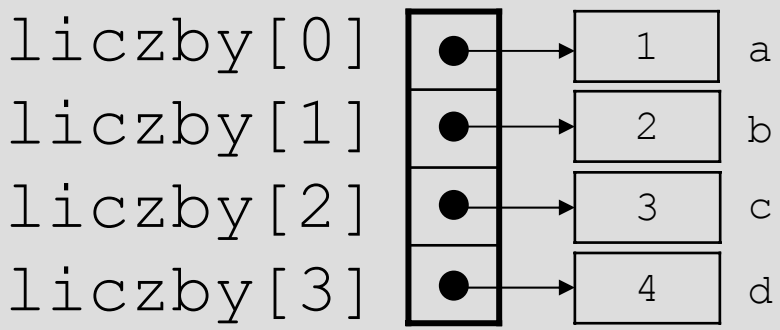
```
int a=1,b=2,c=3,d=4;
int *liczby[4];          //4-elementowa tablica wskaźników do typu int

liczby[0]=&a;
liczby[1]=&b;
liczby[2]=&c;
liczby[3]=&d;

for(int i=0;i<4;i++)
   cout<<*liczby[i]<<" ";          ⟶   1 2 3 4

for(int i=0;i<4;i++)
   cout<<liczby[i]<<" ";           ⟶   0x22ff6c 0x22ff68 0x22ff64 0x22ff60
```

Adresy komórek pamięci, w których są umieszczone zmienne `a`, `b`, `c`, `d`



Napisy, tablice napisów

Napisy

Stała tekstowa (napis, łańcuch znaków, ang. *string*) jest ciągiem znaków. Aby stała tekstowa była stringiem, ciąg ten musi być ujęty w cudzysłów. W języku C++ **napis jest wskaźnikiem do pierwszego swojego znaku**.

Przykład 1:

```
#include <iostream>
#include <cstring>
using namespace std;

main()
{
   string imie_nazwisko;

   cin>>imie_nazwisko;                    <----    Jan Kowalski
   cout<<imie_nazwisko<<endl;             ---->    Jan

   imie_nazwisko="Jan Kowalski";
   cout<<imie_nazwisko<<endl;             ---->    Jan Kowalski

}
```

Napisy

Stała tekstowa (napis, łańcuch znaków, ang. *string*) jest ciągiem znaków. Aby stała tekstowa była stringiem, ciąg ten musi być ujęty w cudzysłów. W języku C++ **napis jest wskaźnikiem do pierwszego swojego znaku**.

Przykład 2:

```
#include <iostream>
#include <cstring>
using namespace std;

main()
{
   char *zdanie="To jest string";
   cout<<zdanie<<endl;                    ⟶   To jest string
   cout<<*zdanie<<endl;                   ⟶   T

   zdanie++;
   cout<<zdanie<<endl;                    ⟶   o jest string
   cout<<*zdanie<<endl;                   ⟶   o

   zdanie--;
   cout<<zdanie<<endl;                    ⟶   To jest string
   cout<<*zdanie<<endl;                   ⟶   T

   zdanie+=3;
   cout<<zdanie[0]<<endl;                 ⟶   j
   cout<<zdanie[1]<<endl;                 ⟶   e
}
```

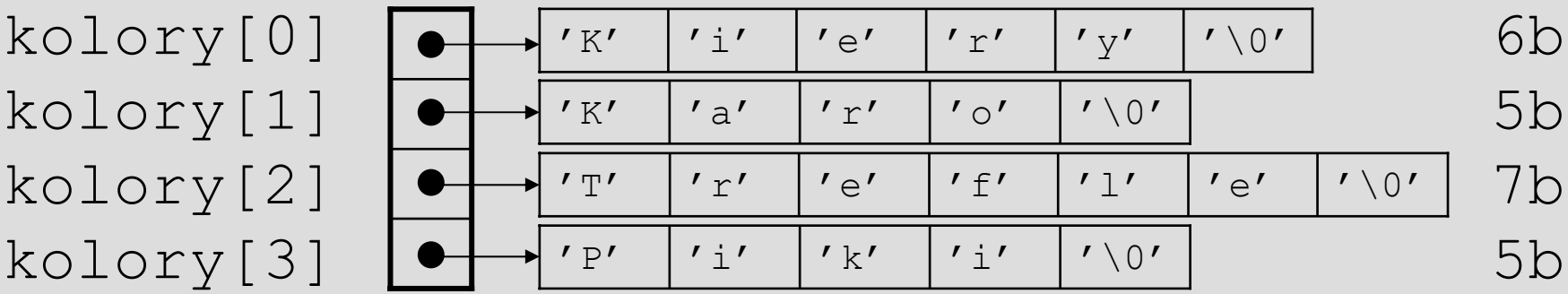
Tablice napisów

W języku C++ **napis jest wskaźnikiem do pierwszego swojego znaku**.

Typowym zastosowaniem **tablicy wskaźników** jest **tablica napisów**. Każdy element takiej tablicy jest napisem, a więc wskaźnikiem do pierwszego znaku napisu.

Przykład:

```
char *kolory[4]={"Kiery","Karo","Trefle","Piki"};
char *kolory[ ]={"Kiery","Karo","Trefle","Piki"};
```



Mimo że w tablicy `kolory` umieszczone są łańcuchy (napisy), znajdują się w niej jedynie wskaźniki. Każdy z nich wskazuje na pierwszy znak odpowiedniego napisu.

Mimo że tablica `kolory` ma stałą długość ([4]), może ona przechowywać napisy o dowolnej
Jest to jeden z przykładów elastyczności i siły struktur zdefiniowanych w języku C++.

Tablice napisów, c.d.

Przykład:

```
char *kolory[4]={”Kiery”,”Karo”,”Trefle”,”Piki”};

for(int i=0;i<4;i++)
   cout<<kolory[i]<<" ";          ⟶      Kiery Karo Trefle Piki

for(int i=0;i<4;i++)
   cout<<*kolory[i]<<" ";         ⟶      K K T P
```

Wskaźniki do funkcji

Wskaźnik do funkcji przechowuje adres funkcji w pamięci operacyjnej komputera.
W języku C++ **nazwa funkcji** (podobnie jak nazwa tablicy) **jest początkowym adresem w pamięci komputera kodu wykonującego jej zadanie**.

Przykład:

```
int funkcja();        //deklaracja funkcji

int (*wskfun)();      //deklaracja wskaźnika do funkcji
wskfun=funkcja;       //ustawienie wskaźnika na funkcję

(*wskfun)();          //wywołanie funkcji za pomocą wskaźnika do funkcji
wskfun();             //wywołanie funkcji za pomocą wskaźnika do funkcji
funkcja();            //wywołanie funkcji za pomocą nazwy funkcji
```

Wskaźniki do funkcji mogą być:
- przekazywane do innych funkcji,
- odbierane jako rezultat wykonania funkcji,
- przypisywane innym wskaźnikom do funkcji.

Kiedy **wskaźniki do funkcji** mogą się przydać?:
- przy przesyłaniu argumentów do innych funkcji – adres funkcji można wysłać jako argument do innej funkcji, która ma u siebie wykonać tę przysłaną funkcję,
- do tworzenia tablic wskaźników do funkcji – w takiej tablicy mamy jakby listę działań (funkcji) do wykonania (patrz następny slajd),

Przykład:

```
////////////////////////////////////////////////////////////////////////
float pole_podstawy(float a,float b)
{
   return a*b;                    //oblicz pole podstawy
}
////////////////////////////////////////////////////////////////////////
float objetosc_prostopadl(float a,float b,float h,float (*wf)(float,float))
{
   return (*wf)(a,b)*h;           //oblicz objętość (wywołanie poprzez wskaźnik)
}
////////////////////////////////////////////////////////////////////////


main()
{
   float bok1=2.4,bok2=3.3,wys=4.0;      //definicja zmiennych

   float (*wskfun)(float,float);         //deklaracja wskaźnika do funkcji

   wskfun=pole_podstawy;                 //ustawienie wskaźnika do funkcji

   cout<<"objetosc="<<objetosc_prostopadl(bok1,bok2,wys,wskfun)<<endl;
}
```

Tablice wskaźników do funkcji

Tablice wskaźników do funkcji

Tablica wskaźników do funkcji przechowuje adresy (wskaźniki) do funkcji.
W takiej tablicy mamy jakby listę działań (funkcji) do wykonania.

Przykład:

```
void funkcja1();                    //deklaracja funkcji
void funkcja2();                    //deklaracja funkcji
void funkcja3();                    //deklaracja funkcji

void (*tabwskfun[3])();             //deklaracja tablicy wskaźników do funkcji

tabwskfun[0]=funkcja1;              //ustawienie elementu tablicy na funkcję 1
tabwskfun[1]=funkcja2;              //ustawienie elementu tablicy na funkcję 2
tabwskfun[2]=funkcja3;              //ustawienie elementu tablicy na funkcję 3

LUB
void (*tabwskfun[3])()={funkcja1,funkcja2,funkcja3};  //definicja tablicy
                                                      //wskaźników do funkcji

(*tabwskfun[0])();     //wywołanie funkcji 1
(*tabwskfun[1])();     //wywołanie funkcji 2
(*tabwskfun[2])();     //wywołanie funkcji 3

tabwskfun[0]();        //wywołanie funkcji 1
tabwskfun[1]();        //wywołanie funkcji 2
tabwskfun[2]();        //wywołanie funkcji 3
```

Typowy błąd programisty

Przekazywanie znaków do funkcji oczekującej podania napisu prowadzi do **krytycznych błędów wykonania programu**.

Typowy błąd programisty

Przekazanie napisu do funkcji oczekującej jednego znaku powoduje **błąd składniowy**.

Typowy błąd programisty

Niewłączenie do programu pliku nagłówkowego `<cstring>`, gdy wykorzystywane są funkcje z biblioteki obsługi napisów.

Wskazówka dotycząca przenośności

Wewnętrzne kody reprezentujące poszczególne znaki mogą być różne w różnych systemach.

Wskazówka dotycząca przenośności

Nie należy jawnie sprawdzać kodu ASCII, np. `if (ch==65)`. Zamiast kodu można wykorzystać odpowiadającą mu stałą znakową, tj. `if (ch=='A')`.

Operacje wejścia/wyjścia

Strumienie

Operacje wejścia/wyjścia (ang. I/O) w C++ występują jako *strumienie* bajtów. Strumień to po prostu sekwencja bajtów. Bajty mogą reprezentować zmaki ASCII, wewnętrzny format surowych danych, obraz graficzny, cyfrową mowę, cyfrowe wideo lub każdy inny rodzaj informacji.

Praca systemowego mechanizmu I/O polega na przesyłaniu bajtów z urządzenia do pamięci głównej i z powrotem w sposób pewny i konsekwentny. W operacjach wejściowych bajty przepływają z urządzenia (np. klawiatury, stacji dysków, połączenia sieciowego) do pamięci głównej. W operacjach wyjściowych bajty przepływają od pamięci głównej do urządzenia (np. ekranu, drukarki, połączenia sieciowego).

Wykonywanie operacji I/O może się odbywać na dwóch poziomach:

- **niskim:** możliwości I/O niskiego poziomu (tj. niesformatowanego) określają typowo, że pewna ilość bajtów powinna być po prostu przesłana z urządzenia do pamięci lub odwrotnie. W takim transferze celem zainteresowania są indywidualne bajty, ale nie jest istotne znaczenie tych bajtów (nie są interpretowane w żaden sposób). Operacje niskiego poziomu umożliwiają szybkie transfery dużej ilości danych, ale nie są specjalnie wygodne dla użytkownika.
- **wysokim:** programiści preferują operacje I/O wysokiego poziomu, czyli formatowane I/O, gdzie bajty grupowane są w jednostki posiadające pewne znaczenie, jak liczby całkowite, liczby zmiennoprzecinkowe, znaki, napisy i typy zdefiniowane przez użytkownika. Polega na przesyłaniu informacji przez strumień łącznie z interpretowaniem jej (formatowaniem). Możliwości te, zorientowane na typy danych, są wystarczające dla większości operacji I/O, poza przetwarzaniem dużej ilości plików.

Program porozumiewa się ze światem zewnętrznym (czyli użytkownikiem siedzącym przed ekranem i klawiaturą, a także z pamięcią zewnętrzną czyli dyskiem) za pomocą operacji wejścia/wyjścia. Operacje wejścia/wyjścia nie są zdefiniowane w samym języku C++, umożliwiają je biblioteki standardowo dołączane przez producentów kompilatorów.

Istnieje kilka bibliotek obsługujących operacje wejścia/wyjścia:

1. Biblioteka **`stdio`** – głównie dla programistów klasycznego C (używanie jej nie należy do dobrego stylu programowania),
2. Biblioteka **`stream`** – jest zbiorem klas zapewniających operacje wejścia/wyjścia (jest starą wersją biblioteki **`iostream`**),
3. Biblioteka **`iostream`** – jest zalecana do stosowania w programowaniu w C++.

UWAGA: W poniższym wykładzie zostaną pokazane tylko niektóre wybrane możliwości biblioteki **`iostream`**, a także istota posługiwania się nią. Aby dokładnie poznać bibliotekę **`iostream`** należy przeczytać jej **opis w dokumentacji danego kompilatora**.

Za wykonywanie operacji I/O na wysokim poziomie odpowiadają klasy:

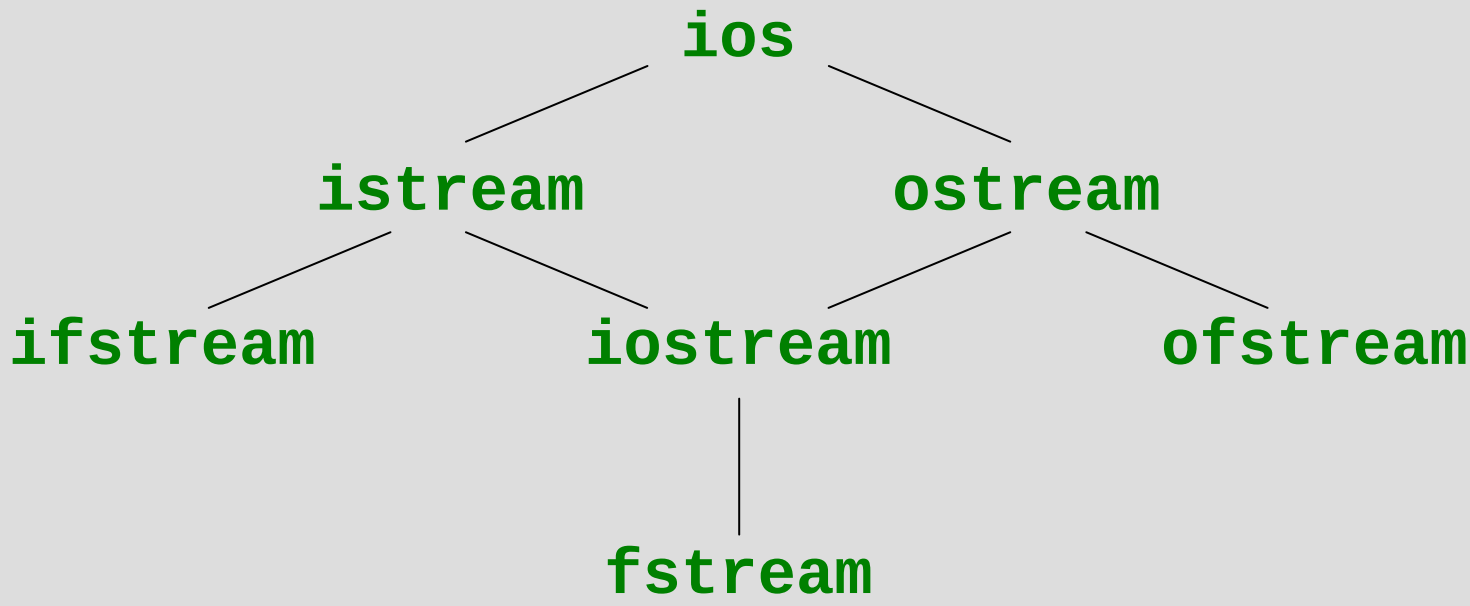
- **`istream`** – strumień wejściowy (wczytujący),
- **`ostream`** – strumień wyjściowy (wypisujący),
- **`iostream`** – klasa, która jest pochodną obu powyższych klas (wielokrotne dziedziczenie). Klasa ta umożliwia definiowanie swoich strumieni mogących wczytywać i wypisywać dane.

Najważniejsze zagadnienia związane z biblioteką `iostream` to:

1. **wyprowadzanie/wprowadzanie informacji ze standardowych urządzeń we/wy** (klawiatura/ekran),
2. **operacje na plikach danych** znajdujących się na nośnikach zewnętrznych (dyskach, taśmach magnetycznych, itp.),
3. **formatowanie wewnętrzne** czyli wpisywanie/odczytywanie informacji w obrębie programu do/z jakiegoś obszaru pamięci (np. do tablicy znakowej `char`).

Jeśli mamy do czynienia z którymś z powyższych zagadnień, należy za pomocą dyrektywy `#include` włączyć do programu odpowiedni plik nagłówkowy zawierający odpowiednie deklaracje:

- `iostream` – jakiekolwiek korzystanie z biblioteki,
- `fstream` – operacje we/wy na plikach zewnętrznych,
- `strstream` – operacje we/wy na tablicach (**formatowanie wewnętrzne**).



Część hierarchii klas strumienia I/O z kluczowymi klasami przetwarzania plików

Strumienie predefiniowane

Kompilator predefiniuje kilka strumieni. Predefiniuje, czyli czynności związane z pracą ze strumieniem są automatycznie wykonane za Ciebie. Innymi słowy: strumień zostanie zdefiniowany i otwarty, można go używać, a po zakończeniu programu strumień zostanie automatycznie zamknięty.

Są cztery predefiniowane strumienie:

`cout` **`cin`** **`cerr`** **`clog`**

Strumienie te to po prostu egzemplarze obiektów pewnych klas. Aby z nich skorzystać należy w programie zamieścić dyrektywę **`#include <iostream>`**.

Znaczenie predefiniowanych strumieni:

- **`cout`**: jest powiązany ze standardowym urządzeniem wyjścia (zwykle ekran),
- **`cin`**: jest powiązany ze standardowym urządzeniem wejścia (zwykle klawiatura),
- **`cerr`**: jest powiązany ze standardowym urządzeniem, na które chce się wypisywać komunikaty o błędach (zwykle ekran). Strumień ten jest niebuforowany,
- **`clog`**: jak wyżej, z tym, że ten strumień jest buforowany.

To, że strumień jest **niebuforowany** oznacza, że jak tylko zażądamy wypisania komunikatu o błędzie, zrobione to zostanie natychmiast.

Buforowanie oznacza tutaj, że robiona jest pewna optymalizacja, polegająca np. na tym, że dopiero gdy zbierze się kilka komunikatów, wtedy zostaną one wpisane do „dziennika pokładowego” (ang. *log-book*) hurtem, oszczędzając czas.

Przykład:

```
#include <iostream>

using namespace std;

main()
{
   int a;
   float b;
   char tekst[40];
   char *zdanie=”Jakies zdanie”;

   cin>>a;                    <---    5 (operator >> wczytywania)
   cout<<a<<endl;             --->    5 (operator << wypisywania)

   cin>>b;                    <---    6.6
   cout<<b<<endl;             --->    6.6

   cin>>tekst;                <---    Dowolny tekst
   cout<<tekst<<endl;         --->    Dowolny

   cout<<zdanie<<endl;        --->    Jakies zdanie
}
```

UWAGA: W przypadku wypisywania na ekran lub wczytywania z klawiatury obowiązują pewne domyślne ustawienia (domniemania). Chcąc zmienić te domniemania należy użyć formatowania.

Formatowanie

Sterowanie formatem

Przy operacjach na strumieniu używane są pewne domniemania dotyczące formatu informacji wstawianej lub wyjmowanej ze strumienia. Jeśli te domniemania nam nie odpowiadają, to możemy je zmienić i od tej pory dany strumień będzie wczytywał (wyjmował) lub wypisywał (wstawiał) według nowych zasad (według nowego formatu). Bieżące zasady formatowania zapisane są w tak zwanych **flagach stanu formatowania**.

Flagi stanu formatowania dotyczące formatu informacji to:
`skipws, left, right, internal, dec, oct, hex, showbase, showpoint, uppercase, showpos, scientific, fixed, unitbuf, stdio`

Flagi te są zdefiniowane w zakresie klasy `ios` i są tam publiczne, zatem można używać ich spoza tego zakresu, poprzedzając je kwalifikatorem zakresu, np.:
`ios::left`
`ios::scientific` itp.

Flagi stanu formatowania umożliwiają ustawianie szerokości pola, precyzji, ustawianie i zerowanie znaczników formatowania, znaków wypełniających pola, opróżnianie strumienia, wstawianie do strumienia znaków nowego wiersza i opróżnianie strumienia, wstawianie do strumienia wyjściowego znaku zerowego (NULL) oraz pomijanie odstępów w strumieniu wejściowym.

UWAGA: Informacja zapisana w flagach jest typu logicznego: tak/nie (prawda/fałsz) (np. pokazywać kropkę dziesiętną: tak czy nie?)

Znaczenie poszczególnych **flag stanu formatowania**:

`skipws`

przeskakuj białe znaki (`ios::skipws`)

ustawienie tej flagi powoduje ignorowanie w procesie wyjmowania ze strumienia ewentualnych białych znaków (spacje, tabulatory, znaki nowej linii, itp.), które poprzedzają oczekiwane znaki.

Domniemanie: flaga jest ustawiona.

Przykład: *`[spacja][tabulator][spacja]`*`247`

`left`
`right`
`internal`

pole justowania (`ios::left`) (`ios::right`) (`ios::internal`)

justowanie polega na tym, że podczas wypisywania liczby może być ona dosunięta do lewej (`left`) lub prawej (`right`) krawędzi obszaru albo wypisana tak, że ewentualny znak dosunięty jest do lewej, a liczba do prawej krawędzi obszaru, a wewnątrz (`internal`) są znaki wypełniające.

Domniemanie: ustawiona jest flaga `right`.

Przykład:

```
-25_________        left
_________-25        right
-_________25        internal
```

Znaczenie poszczególnych **flag stanu formatowania**:

`dec`
`oct`
`hex`

pole podstawy konwersji (`ios::basefield`)

określanie w jakim systemie (zapisie) wczytywane lub wypisywane są liczby całkowite. Podstawą konwersji dla liczb całkowitych może być liczba:

`10` - konwersja dziesiątkowa (decymalna)
`8` - konwersja ósemkowa (oktalna)
`16` - konwersja szesnastkowa (heksadecymalna)

Domniemanie: ustawiona jest flaga `dec`.

`showbase`

pokaż podstawę (`ios::showbase`)

ustawienie tej flagi jest żądaniem by liczby wypisywane były tak, żeby łatwo można było rozpoznawać w jakim są systemie. Zatem przed liczbą szesnastkową staną znaki 0x, przed liczbą ósemkową 0, przed liczbą dziesiątkową nic.

Domniemanie: flaga nie jest ustawiona.

Przykład:

	flaga ios::showbase ustawiona	nie ustawiona
hex	0xa4c	a4c
oct	077	77
dec	32	32

Znaczenie poszczególnych **flag stanu formatowania**:

`showpos` **pokaż dodatnie (`ios::showpos`)**

ustawienie tej flagi powoduje, że przy wypisywaniu dodatnich liczb dziesiątkowych zostaną one poprzedzone znakiem + (plus).

Domniemanie: flaga nie jest ustawiona.

Przykład:

```
flaga ios::showpos
ustawiona            nie ustawiona
+204.2               204.2
```

`uppercase` **wielkie litery (`ios::uppercase`)**

w zapisie niektórych typów liczb występują litery oznaczające np. podstawę konwersji `'x'` lub wykładnik `'e'` w notacji naukowej. Ustawienie tej flagi powoduje, że litery te są wypisywane jako wielkie `X` lub `E`.

Domniemanie: flaga nie jest ustawiona.

Przykład:

```
flaga ios::uppercase
ustawiona            nie ustawiona
0Xa4c                0xa4c
44E-6                44e-6
```

Znaczenie poszczególnych **flag stanu formatowania**:

`showpoint` **pokaż kropkę dziesiętną (`ios::showpoint`)**

ustawienie tej flagi powoduje, że przy wypisywaniu liczb zmiennoprzecinkowych wypisywane są nawet nieznaczące zera i kropka dziesiętna.

Domniemanie: flaga nie jest ustawiona.

Przykład:

```
flaga ios::showpoint
ustawiona              nie ustawiona
2.34000                2.34
3.00000                3
```

`scientific`
`fixed` **notacja liczby zmiennoprzecinkowych (naukowa/zwykła) (`ios::scientific`) (`ios::fixed`)**

ustawienie flagi `scientific` lub `fixed` sprawia, że liczby będą wypisywane odpowiednio w tzw. notacji naukowej (czyli wykładniczej) lub w zwykłej notacji (czyli dziesiętnej [nie dziesiątkowej, ale dziesiętnej]).

Domniemanie: jeśli flaga nie jest ustawiona, sposób wypisywania liczby zależy od samej liczby.

Przykład:

```
ios::scientific      ios::fixed
2.5e3                2500
```

Znaczenie poszczególnych **flag stanu formatowania**:

`unitbuf` **(`ios::unitbuf`)**

ustawienie tej flagi jest rezygnacją z tak zwanego buforowania strumienia. Strumień niebuforowany nie jest tak efektywny, jak buforowany.
Domniemanie: flaga jest ustawiona.

`stdio` **(`ios::stdio`)**

ustawienie tej flagi jest żądaniem, by po każdym wstawieniu czegoś do strumienia `stdio` i `stderr` strumień popłynął od razu i informacja bez zwłoki pojawiła się na ekranie.

Sposoby zmiany trybu formatowania

Jeśli chcemy zmienić format wypisywania informacji na ekranie przez strumień wyjściowy `cout`, lub format wczytywania informacji z klawiatury strumieniem wejściowym `cin`, możemy posłużyć się funkcjami składowymi klasy `ios`, i za ich pomocą możemy posługiwać się **flagami stanu formatowania**.

Flagi stanu formatowania możemy zmodyfikować za pomocą:

1. **elementarnych funkcji składowych** klasy `ios`, czyli funkcji `setf`, `unsetf` – sposób ten wymaga jednak pamiętania nazw wszystkich omówionych wcześniej flag formatowania.
2. **„wygodnych" funkcji składowych** klasy `ios`, ale takich, których nazwy same przypominają to, co robią – każda z takich funkcji służy do modyfikacji jednej określonej flagi.
3. **„bardzo wygodnych" manipulatorów** – zamiast wywoływać funkcję składową dla danego strumienia, „wpuszczamy do niego" specjalne kody, które strumień zinterpretuje jako życzenie zmiany sposobu formatowania.

Niektóre z funkcji (prototypy) dotyczące zmiany formatu informacji to:

```
long setf(long flaga,long nazwa_pola);          //ustaw wyszczególnione flagi
long setf(long flaga);                          //ustaw wyszczególnione flagi
long unsetf(long flaga);                        //wyzeruj wyszczególnione flagi
long flags(long wzor);                          //ustaw wszystkie flagi wg wzoru
long flags(void);                               //zwróć bieżące ustawienia flag
```

lub bardziej wygodne

```
int width(int);                  //ustawienie szerokości pola wypisania liczby
int precision(int);              //ustawienie precyzji wyświetlania
int fill(int);                   //wypełnianie pola wypisania dowolnym znakiem
```

UWAGA: jest kilka sposobów zrobienia tego samego, tj. zmiany trybu formatowania.

Zmiana trybu formatowania za pomocą elementarnych funkcji `setf` i `unsetf`

```
#include <iostream>
using namespace std;

main()
{
   float a=2345,b=-4.56;

   cout<<a<<"\t\t"<<b<<endl;                  ──►  2345                 -4.56

   cout.setf(ios::showpoint);
   cout<<a<<"\t\t"<<b<<endl;                  ──►  2345.00              -4.56000

   cout.setf(ios::scientific);
   cout<<a<<"\t\t"<<b<<endl;                  ──►  2.345000e+003        -4.560000e+000

   cout.setf(ios::uppercase);
   cout<<a<<"\t\t"<<b<<endl;                  ──►  2.345000E+003        -4.560000E+000

   cout.unsetf(ios::scientific);
   cout<<a<<"\t\t"<<b<<endl;                  ──►  2345.00              -4.56000

   cout.unsetf(ios::showpoint);
   cout<<a<<"\t\t"<<b<<endl;                  ──►  2345                 -4.56

   cout.setf(ios::showpos);
   cout<<a<<"\t\t"<<b<<endl;                  ──►  +2345                -4.56

}
```

Zmiana trybu formatowania za pomocą „wygodnych” funkcji składowych

```
#include <iostream>
using namespace std;

main()
{
   float a=2345,b=-4.56,c=6;

   cout<<a<<"\t\t"<<b<<endl;                  →  2345                -4.56

   cout.width(6);
   cout<<a<<"\t\t"<<b<<endl;                  →  __2345              -4.56 (dot. najbliższej operacji I/O)

   cout.width(6);
   cout<<a<<"\t\t";
   cout.width(6);
   cout<<b<<endl;                             →  __2345              _-4.56

   cout<<a;
   cout.width(17);
   cout.fill('*');
   cout<<b<<endl;                             →  2345***************-4.56

   a=2345.6789012345;
   cout<<a<<"\t\t"<<b<<"\t\t"<<c<<endl;       2345.68             -4.56              6

   cout.precision(10);
   cout<<a<<"\t\t"<<b<<"\t\t"<<c<<endl;       2345.678955         -4.559999943       6

   cout.setf(ios::showpoint);
   cout<<a<<"\t\t"<<b<<endl;                  2345.678955         -4.559999943       6.000000000

}
```

Zmiana trybu formatowania za pomocą „wygodnych” manipulatorów

```
#include <iostream>
#include <iomanip>
using namespace std;

main()
{
   int a=45;

   cout<<a<<” ”<<hex<<a<<” ”<<oct<<a<<” ”<<dec<<a<<endl;          ⟶   45 2d 55 45

   cout<<setw(5)<<a<<setw(3)<<a<<setw(4)<<a<<endl;               ⟶   ___45_45__45

   cout<<a<<setfill('*')<<setw(5)<<a<<setfill('*')<<
   setw(5)<<a<<endl;                                             ⟶   45***45***45

   float b=45.6789;

   cout<<b<<" "<<setprecision(4)<<b<<endl;                       ⟶   45.6789 45.68

   cout<<hex<<a<<” ”<<setiosflags(ios::uppercase)<<
   a<<” ”<<resetiosflags(ios::uppercase)<<a<<endl;               ⟶   2d 2D 2d

   cout<<dec<<a<<” ”<<setiosflags(ios::uppercase | ios::showpos)<<
   a<<” ”<<resetiosflags(ios::uppercase | ios::showpos)<<a<<endl;    45 +45 45

}
```

Typowy błąd programisty

Próba odczytu z `ostream` (lub innego, tylko wyjściowego, strumienia) oraz zapisu do `istream` (lub innego, tylko wejściowego, strumienia) jest błędem.

Typowy błąd programisty

Założenia, że ustawienie szerokości pola dotyczy wszystkich wyjść, a nie tylko następnego (dla wysyłania lub wprowadzania) jest błędem.

Typowy błąd programisty

Jeśli nie dostarczymy wystarczająco szerokiego pola do obsługi wyjścia, wydruk będzie miał taki rozmiar, jaki potrzeba. Może to jednak spowodować, że będzie on nieczytelny i trudny do odczytania.

Dobry styl programisty

W programach C++ stosuj wyłącznie formę C++ operacji I/O, mimo że dostępny jest też sposób C tych operacji.

Wskazówka dotycząca wydajności

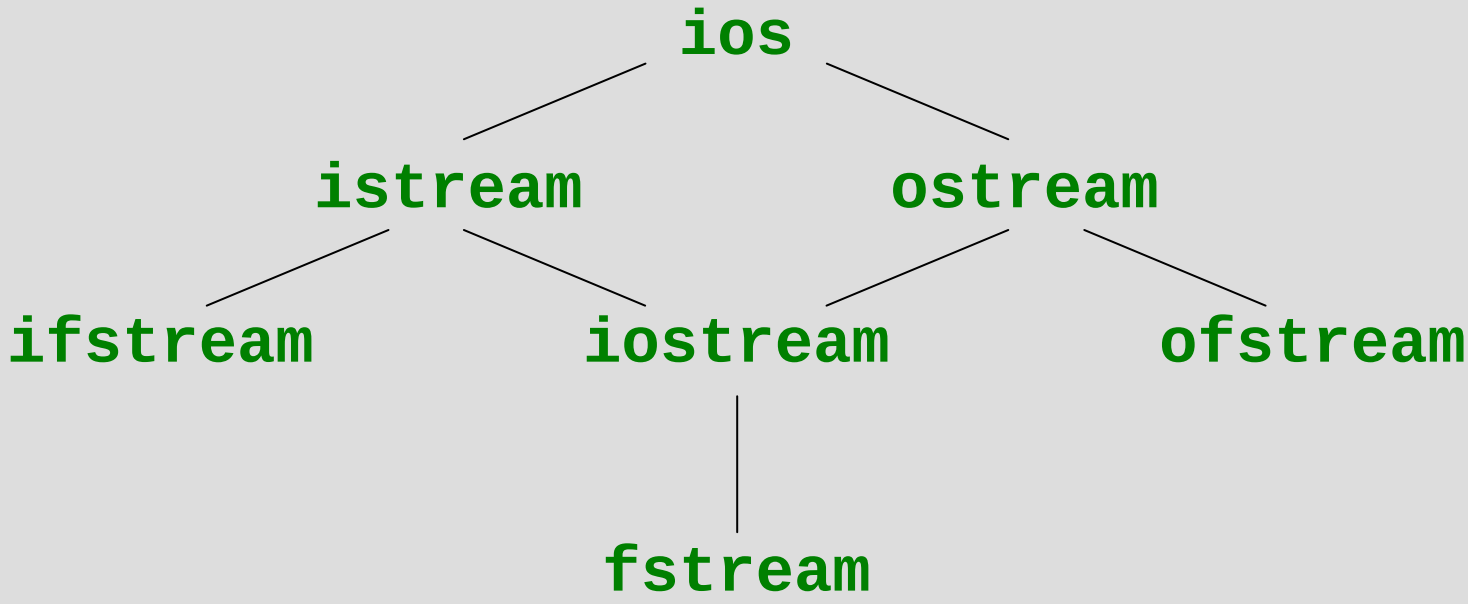
W celu uzyskania większej wydajności przy przetwarzaniu dużej ilości plików stosuj nieformatowane operacja I/O, zamiast formatowane.

Wskazówka praktyczna

C++ pozwala na wspólne traktowanie operacji I/O dla typów predefiniowanych, jak i określonych przez użytkownika.

Operacje na plikach dyskowych

Jeśli chcemy wykonać przetwarzanie plików w C++, należy włączyć pliki nagłówkowe **<iostream>** oraz **<fstream>**. Pliki są otwierane przez tworzenie obiektów klas strumieni. Dzięki temu, wszystkie funkcje składowe, operatory i manipulatory opisane wcześniej (podczas omawiania operacji wejścia/wyjścia i formatowania) mogą być również stosowane do strumieni plików.



Część hierarchii klas strumienia I/O z kluczowymi klasami przetwarzania plików

Zapis do pliku

```
#include <iostream>
#include <fstream>

using namespace std;

main()
{
   ofstream zapis(”klienci.dat”,ios::out);      //otwarcie pliku w celu zapisu
   //ofstream zapis(”klienci.dat”);             //domyślnie obiekt ofstream
                                                //jest otwierany do zapisu

   cout<<”Wprowadz nr konta, nazwisko klienta i bilans.”<<endl;
   cout<<”Wpisz znak konca pliku, by zakonczyc wprowadzanie.”<<endl;

   int nr_konta;
   char nazwisko[50];
   float bilans;

   while(cin>>nr_konta>>nazwisko>>bilans)
   {
      zapis<<nr_konta<<’ ’<<nazwisko<<’ ’<<bilans<<endl;
   }

   zapis.close();                               //zamknięcie pliku
}
```

Odczyt z pliku

```
#include <iostream>
#include <fstream>

using namespace std;

main()
{
   ifstream odczyt(”klienci.dat”,ios::in);      //otwarcie pliku do odczytu
   //ifstream odczyt(”klienci.dat”);            //domyślnie obiekt ifstream
                                                //jest otwierany do odczytu

   int nr_konta;
   char nazwisko[50];
   float bilans;

   while(odczyt>>nr_konta>>nazwisko>>bilans)
   {
      cout<<nr_konta<<’ ’<<nazwisko<<’ ’<<bilans<<endl;
   }

   odczyt.close();                              //zamknięcie pliku
}
```

Tryby otwarcia pliku

TRYB	OPIS
`ios::app`	dopisz do końca pliku
`ios::ate`	otwórz plik do zapisu i przesuń na koniec pliku (zwykle używany do dodawania danych do końca pliku); dane mogą być zapisane gdziekolwiek w pliku
`ios::in`	otwórz plik do odczytu
`ios::out`	otwórz plik do zapisu (usuń jego zawartość)
`ios::trunc`	jeśli plik istnieje, usuń jego zawartość (jest to także domyślne działanie **`ios::out`**)
`ios::nocreate`	operacja otwarcia nie powiedzie się, jeśli plik nie istnieje
`ios::noreplace`	operacja otwarcia nie powiedzie się, jeśli plik istnieje

Kombinacja klawiszy oznaczająca koniec pliku dla różnych systemów komputerowych

SYSTEM KOMPUTEROWY	KOMBINACJA KLAWISZY
IBM PC i kompatybilny	**<ctrl>z**
VAX(VMS)	**<ctrl>z**
UNIX	**<ctrl>d**
MACINTOSH	**<ctrl>d**

Typowy błąd programisty

Otwarcie istniejącego pliku do zapisu (`ios::out`), gdy w rzeczywistości użytkownik chce go zachować. Zawartość pliku jest wtedy usuwana bez ostrzeżenia.

Typowy błąd programisty

Zastosowanie niewłaściwego obiektu `ofstream` przy odwołaniu się do pliku.

Typowy błąd programisty

Nieotwarcie pliku przed próba odwołania się do niego w programie.

Dobry styl programisty

Jeśli zawartość pliku nie powinna być modyfikowana, należy otworzyć plik w trybie tylko do odczytu stosując `ios::in`. Zabezpiecza to przed niezamierzoną modyfikacją jego zawartości.

Wskazówka dotycząca wydajności

Jawnie zamykaj każdy plik, gdy tylko wiesz, że program nie będzie się do niego ponownie odwoływał. Może to zmniejszyć wykorzystanie zasobów w programie, a także zwiększa przejrzystość programu.

PODSUMOWANIE:

- Możliwe jest tworzenie tablic wskaźników. Tablice te zawierają adresy do pewnych obszarów pamięci.
- Wskaźnik do funkcji to adres, pod którym znajduje się kod tej funkcji.
- Wskaźniki do funkcji mogą być przekazywane do innych funkcji, zwracane przez nie, przechowywane w tablicach oraz przypisywane innym wskaźnikom.
- Typowe zastosowanie wskaźników do funkcji ma miejsce w systemach sterowanych opcjami wybieranymi z menu. Są one wykorzystywane do określania, która funkcja ma zostać wywołana po wybraniu z menu określonej opcji.
- Operator `new` automatycznie tworzy obiekt o odpowiedniej wielkości, wywołuje jego konstruktor, a następnie zwraca do niego wskaźnik. Pamięć zajmowana przez tak utworzony obiekt może być zwolniona wyłącznie operatorem `delete`.
- Operacje I/O wykonywane są w sposób „czuły” na typ danych.
- Operacje I/O w C++ następują w strumieniach bajtów. Strumień to po prostu sekwencja bajtów.
- C++ umożliwia dokonywanie „wysokopoziomowych” i „niskopoziomowych” operacji I/O. Operacje niskopoziomowe przesyłają pewną ilość bajtów z urządzenia do pamięci i odwrotnie. Operacje wysokopoziomowe wykonywane są na bajtach zgrupowanych w jednostki mające znaczenie, jak liczby całkowite, liczby zmiennoprzecinkowe, znaki, napisy i typy zdefiniowane przez użytkownika.
- Większość programów C++ włącza plik nagłówkowy `<iostream>` zawierający podstawowe informacje wymagane dla wszystkich strumieniowych operacji I/O. Klasa ta obsługuje operacje I/O strumienia.
- Nagłówek `<iomanip>` zawiera informacje niezbędne do formatowanego wejścia/wyjścia z parametryzowanymi manipulatorami strumienia.
- Nagłówek `<fstream>` zawiera informacje potrzebne do operacji przetwarzania plików.

Następny wykład

Wykład nr 5

Temat: Klasy i abstrakcja danych, cz. I.