

Wejście-wyjście

- Nie jest elementem języka C++
- Niezbyt istotne dla użytkownika, ważne dla języka:
 - jest mocny, skoro da się w nim wyrazić nowe pojęcia;
 - łatwiej przenieść go na nową platformę
 - jest mniejszy.
- Standardowa biblioteka we-wy `iostream`.
- Operacje we-wy wykonuje się na *strumieniach*.

Strumienie

- Strumienie są ciągami znaków.
- Operacje we-wy polegają na przekształcaniu obiektów na ciągi znaków i vice versa.
- Zaleta strumieni: wygodne i jednolite traktowanie typów wbudowanych i zdefiniowanych przez użytkownika.

Wyjście

- Standardowe strumienie: `cout` i `cerr`.
- `cout` odpowiada `stdout` z C a `cerr` — `stderr`.
- Ponadto: `clog` (jak `cerr`, ale buforowany)
- Wszystkie mają swoje odmiany dla Unicode: `wcout`,...

Operator «

`ostream& operator<<(ostream&, T)`

- T może być (i często jest) typem referencyjnym.
- typ wyniku umożliwia skróty takie jak:

```
cout << "i=" << i << '\n' ;
```

zamiast

```
cout<<"i="; cout<<i; cout<<'\n';
```

- Operatory `<< i >>` wiążą w lewo. Priorytet `<<` jest niższy niż operacji arytmetycznych. Dzięki temu można napisać np.

```
cout << "a*x+b"= << a*x+b ;
```

- Są jednak operatory o jeszcze niższych priorytetach:

```
cout << "x^y|z=" << x^y|z ; // Błąd!
```

```
cout << "x^y|z=" << (x^y|z) ; // Tak!
```

Operator «

- W treści operatora należy jako wynik przekazać strumień będący parametrem operatora.
- Ponieważ pierwszy argument pochodzi z klasy ostream, nie można go zdefiniować jako metodę klasy T, lecz tylko jako funkcję.
- Zwykle powinna to być funkcja zaprzyjaźniona z klasą T.
- Zamiast operatora << można dla znaków używać funkcji put, np.

```
cout.put( 'A' )
```

zaś dla napisów funkcji write.

Przykład

```
class Zespólone {  
    double re, im ;  
public:  
    friend ostream&  
        operator<<(ostream& os, Zespólone& z);  
}  
ostream& operator<<(ostream& os, Zespólone& z)  
{  
    os<<'('z.re<<', '<<z.im<<")";  
    return os ;; // Bardzo ważne!  
}
```

Klasa ostream

W uproszczeniu:

```
class ostream: public virtual ios{
//...
public:
    ostream& operator<<(char);
    ostream& operator<<(const char*);
    ostream& operator<<(int);
    //...
    ostream& operator<<(ostream& (*pf)(ostream &));
    ostream& put(char);
    ostream& write(const char *);
    ostream& flush() ;
};
```

Wyjście

- Standardowy strumień `cin`.

- Operator `>>`

`istream& operator>>(istream&, T)`

zwykle implementujemy go tak:

```
istream& operator>>(istream& is, T& zmienna)
{
    // pomiń białe znaki: spacja, tab, nowa linia...
    // wczytaj tekstową reprezentację T na zmienną
    return is ;
}
```

- Uwaga: pamiętajmy, że strumienie wyjściowe mają typ `ostream`, a wyjściowe `istream`. Istnieje również typ mieszany `iostream`.

Czy udało się odczytać?

- Zapis

```
if(s>>zmienna) ...
```

jest poprawny i oznacza sprawdzenie, czy wczytanie zmiennej się powiodło.

- Taki zapis jest możliwy dzięki operatorowi konwersji typu `istream` do `int`.

Przykład:

```
main()  
{  
    int i;  
    while(cin>>i) cout<<i ;  
}
```

Metody get

Zamiast operatora >> możemy użyć jednej z metod get:

```
char buf[100];  
cin >> buf; // Tu grozi nam wyjście poza tablicę  
cin.get(buf, 100); // Tu jesteśmy bezpieczni
```

```
class istream : public virtual ios { // ...  
    istream& get(char* s, int n, char delim='\n');  
    istream& get(char c);
```

- `cin<<c` pomija białe znaki, `cin.get(c)` nie.
- `cin>>buf` wczyta tylko do pierwszego białego znaku, `cin.get(buf)` do pierwszego wystąpienia `delim` (domyślnie koniec linii).

Funkcje pomocnicze

- Nagłówek `cctype.h` zawiera użyteczne funkcje klasyfikacji znaków, np.
 - `isspace(char)` — czy biały znak
 - `isalpha` — czy litera
 - `isdigit` — czy cyfra
 - ...i wiele innych
- Metoda `putback(char)` pozwala na wstawienie znaku z powrotem do strumienia (ale ostrożnie).
- Metoda `peek()` pozwala na podejrzenie następnego znaku, bez jego pobierania.

Stany strumienia

- Z każdym strumieniem związany jest jego stan.
- Klasa podstawowa strumieni `ios` definiuje metody testowania stanu, np.

```
int eof() ; // napotkano koniec strumienia
int good(); // operacja się udała
int fail(); // operacja się nie udała
int bad() ; // strumień jest do niczego
```

Bity stanu

- W rzeczywistości stan strumienia jest reprezentowany przez ciąg bitów, które można ustawiać, zerować i testować:

```
int s = cin.rdstate(); // odczytaj bity stanu
if(s & ios::goodbit){ // operacja się udała
} else if (s & ios::failbit){ // niedobrze
} else if (s & ios::badbit){ // pacjent nie przeżył
}
cin.clear(ios::badbit) // wbrew nazwie ustawia badbit
```

Przykład

```
istream& operator>>(istream& s, Zespólona& z)
{
    double re=0, im=0;
    char c = 0;
    c << c;
    if(c=='(') {
        s>>re>>c;
        if(c==',') s>>im>>c;
        if(c!=')') s.clear(ios::badbit);
    }
    else {s.putback(c);s>>re;}
    if(s) z=Zespólona(re,im);
    return s;
}
```

Formatowanie

- Klasa `ios` zawiera operacje służące do określania sposobu wypisywania danych, np.

```
int width(int w);
```

określa minimalną szerokość pola, na którym będzie wypisywany następny element, np.

```
cout << '(' ;  
cout.width(4);  
cout << 12 << ')';
```

Spowoduje wypisanie 12 w polu szerokości 4: (12)

- Uwaga: `width` dotyczy tylko jednej, następnej operacji.
- Domyślnie jest 0, czyli (tylko) tyle znaków ile potrzeba.

Manipulatory

- Manipulatory są pojedynczymi elementami wstawianymi do strumienia, wpływającymi na realizację wyjścia.
- Poznaliśmy już manipulator `endl`:

```
cout << "Kaczka dziwaczka" << endl ;
```

Powoduje on przejście do nowej linii i opróżnienie bufora, czyli

```
cout << '\n' ; cout.flush() ;
```

Zamiast `width` można użyć manipulatora:

```
cout << '(' << setw(4) << 12 << ')';
```


Implementacja manipulatorów

Dzięki przeciążaniu, tworzenie własnych manipulatorów nie jest trudne:

```
ostream& flush(ostream&);  
class Flushtype {} ; // potrzebujemy nowego typu  
  
ostream& operator<<(ostream& os, Flushtype)  
{  
    return flush(os) ;  
}  
Flushtype FLUSH;
```

Teraz możemy wymuszać opróżnienie bufora przez wysłanie FLUSH:

```
cout << x << FLUSH << y << FLUSH;
```

Lepsza implementacja

Powyższą metodę można uogólnić na wszystkie funkcje takiego typu jak `flush`:

```
typedef ostream& (*manip)(ostream&);
```

czyli `manip` jest wskaźnikiem do funkcji o zarówno argumencie jak i wyniku typu `ostream&`

```
ostream& operator<<(ostream& os, manip f)
{
    return f(os) ;
}
```

Teraz śmiało możemy napisać

```
cout << flush ;
```

W rzeczywistości klasa `ostream` ma już odpowiednie metody.

Strumienie i pliki

- Mechanizmu strumieni można używać dla wejścia wyjścia plikowego.
- Należy w tym celu włączyć nagłówek `<fstream>`
- Plik otwiera się dla wyjścia przez utworzenie obiektu klasy `ofstream` z nazwą pliku jako argumentem.
- Podobnie dla wejścia, tyle, że używamy `ifstream`.
- W obu wypadkach warto sprawdzić stan strumienia, y przekonać, że udało się otworzyć plik.

Przykład

```
int main(int argc, char* argv[]){
    ifstream skad(argv[1]);
    if !(skad) cerr << "Bład:...";

    ofstream dokad(argv[2]);
    if !(dokad) cerr << "Bład:...";

    char ch;
    while(skad.get(ch)) dokad.put(ch);

    if(!skad.eof() || dokad.bad()) cerr << "Bład:...";
}
```

Zamykanie strumieni

- Strumień związany z plikiem powinien być na końcu zamknięty.
- Zwykle nie musimy się o to martwić - załatwia to destruktory strumienia.
- Jeśli jednak potrzeba zamknąć strumień przed końcem jego zasięgu, można to uczynić metodą `close()`
- Oczywiście zamkniętego strumienia nie można już używać.

Jeszcze o napisach

- Do tej pory do przechowywania napisów używaliśmy tablic.
- Wadą tablic jest ich stały rozmiar
- Mechanizm klas pozwala na lepsze rozwiązania.
- Biblioteka standardowa C++ zawiera klasę `string`, implementującą ciągi składające się ze zmiennej liczby znaków.
- By jej używać, należy włączyć nagłówek `<string>`

Konstruowanie napisów

```
string x0;           // pusty napis
string x2('a');      // "a"
string x3('a',5);    // "aaaaa"
string x4("abcde");  // "abcde"
string x4("abcdef",5); // "abcde"
string x6(x5);       // "abcde"
string x7(x5,1);     // "bcde"
string x8(x5,0,3);   // "abc"
```

Wycinanie

`x0[1]` // drugi znak `x0`

`x0.substr()` // kopia całości

`x0.substr(pos)` // od pozycji `pos`

`x0.substr(pos,n)` // max `n` znaków od pozycji `pos`

`x0.remove(pos)` // usuń od pozycji `pos`

`x0.remove(pos,n)` // max `n` znaków od pozycji `pos`

Sklejanie

```
string stringOne("Hello");  
    string stringTwo("World");  
  
stringOne += " " + stringTwo;  
  
stringOne = "hello";  
stringOne.append(" world");  
  
string stringThree("Hello");  
                                // doklej " world":  
stringThree.append(stringOne, 5, 6);  
cout << stringThree << endl;
```

Porównywanie i wyszukiwanie

```
x0 == x1
```

```
x0 < x1 // Można używać relacji
```

```
x0.find(c) // znajdź znak
```

```
x0.find(s) // znajdź napis
```

```
x0.find(s,m,n) // znajdź w wyznaczonym fragmencie
```

```
x0.rfind(...) // to co find, tylko od tyłu
```

Oczywiście napisy współpracują ze strumieniami:

```
string x;
```

```
cin >> x;
```

```
cout << x;
```

```
getline(cin,x0); // Dlaczego nie cin.getline()?
```