Dzisiejszy wykład

- Napisy
- Strumienie wejścia-wyjścia

Napisy jako ciągi znaków

- Napis jest ciągiem znaków
- Biblioteka standardowa C++ dostarcza operacje do obsługi napisów
  - indeksowanie
  - przypisanie
  - porównanie
  - dołączanie
  - konkatenacja
  - wyszukiwanie podnapisów
- Znaki są w pamięci komputera przechowywane w postaci liczb
  - Wiele możliwych zbiorów znaków
  - Różne standardy 8-bitowe (Latin1, Latin2,...)
  - Znaki 16 i 32-bitowe (Unicode)

Trejty znakowe

- W standardowym napisie *string* wymaga się, by typ znakowy nie miał operacji kopiowania zdefiniowanej przez użytkownika
  - poprawia to efektywność i upraszcza implementację
  - ułatwia realizację operacji wejścia-wyjścia
- Właściwości typu znaku definiuje jego *char_traits*, jest to specjalizacja wzorca

```
template<class Ch> struct char_traits{ };
```

- Wszystkie *char_traits* są zdefiniowane w *std*, a standardowe trejty udostępnia nagłówek *<string>*

Trejty znakowe

```
template<> struct char_traits<char> {
 typedef char char_type; // type of character
 static void assign(char_type&, const char_type&) ; // = for char_type
 // integer representation of characters:
 typedef int int_type; // type of integer value of character
 static char_type to_char_type(const int_type&) ; // int to char conversion
 static int_type to_int_type(const char_type&) ; // char to int conversion
 static bool eq_int_type(const int_type&, const int_type&) ; // ==
 // char_type comparisons:
 static bool eq(const char_type&, const char_type&) ; // ==
 static bool lt(const char_type&, const char_type&) ; // <
 // operations on s[n] arrays:
 static char_type*move(char_type* s, const char_type* s2, size_t n) ;
 static char_type* copy(char_type* s, const char_type* s2, size_t n) ;
 static char_type* assign(char_type* s, size_t n, char_type a) ;
 static int compare(const char_type* s, const char_type* s2, size_t n) ;
 static size_t length(const char_type*) ;
 static const char_type* find(const char_type* s, int n, const char_type&) ;
 // I/O related:
 typedef streamoff off_type; // offset in stream
 typedef streampos pos_type; // position in stream
 typedef mbstate_t state_type; // multibyte stream state
 static int_type eof() ; // endoffile
 static int_type not_eof(const int_type& i) ; // i unless i equals eof();
                                              // if not any value!=eof()
 static state_type get_state(pos_type p) ;
                                  // multibyte conversion state of character in p
};
```

Trejty znakowe

- Implementacja standardowego wzorca napisowego, *basic_string*, korzysta z tych typów i funkcji
- Typ znaku użyty w *basic_string* musi zapewniać specjalizację *char_traits*, która dostarcza je wszystkie
- Żeby znak mógł pełnić funkcję *char_type*, musi być możliwe uzyskanie wartości całkowitej odpowiadającej każdemu znakowi. Typem tej wartości jest *int_type*. Konwersji między nim a typem znaku dokonuje się za pomocą *to_char_type()* i *to_int_type()*. Dla *char* konwersja jest elementarna
- Zarówno *move(s, s2, n)* jak i *copy(s, s2, n)* kopiują *n* znaków z *s2* do *s*. *move()* działa poprawnie nawet gdy łańcuchy się nakładają, *copy()* może być więc szybsze.
- Wywołanie *assign(s, n, x)* wpisuje *n* kopii *x* do *s*.
- Funkcja *compare()* do porównywania znaków używa *lt()* i *eq()* i przekazuje wartość typu *int*, konwencja wartości zwracanej jest taka sama, jak w *strcmp()*
- Szeroki znak, tzn. obiekt typu *wchar_t* jest podobny do *char*, ale zajmuje dwa bajty lub więcej. Właściwości *wchar_t* opisuje *char_traits<wchar_t>*

```
template<> struct char_traits<wchar_t> {
 typedef wchar_t char_type;
 typedef wint_t int_type;
 typedef wstreamoff off_type;
 typedef wstreampos pos_type;
 // like char_traits<char>
};
```

basic_string

- Podstawą udogodnień napisowych z biblioteki standardowej jest wzorzec *basic_string*, który zapewnia typy składowe i operacje podobne do dostarczanych przez kolekcje standardowe

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class std::basic_string {
 public:
 // ...
};
```

- Wzorzec ten i związane z nim udogodnienia są zdefiniowane w przestrzeni nazw *std* i udostępniane przez nagłówek <*string*>
- Dwie definicje typów dostarczają konwencjonalne nazwy dla popularnych typów napisowych

```
typedef basic_string<char> string;
typedef basic_string<wchar_t>wstring;
```

Typy

- basic_string udostępnia spokrewnione ze sobą typy przez zbiór nazw typów składowych

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // types
 typedef Tr traits_type; // specific to basic_string
 typedef typename Tr::char_type value_type;
 typedef A allocator_type;
 typedef typename A::size_type size_type;
 typedef typename A::difference_type difference_type;
 typedef typename A::reference reference;
 typedef typename A::const_reference const_reference;
 typedef typename A::pointer pointer;
 typedef typename A::const_pointer const_pointer;
 typedef implementation_defined iterator;
 typedef implementation_defined const_iterator;
 typedef std::reverse_iterator<iterator> reverse_iterator;
 typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
 // ...
};
```

Iteratory

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // iterators
 iterator begin() ;
 const_iterator begin() const;
 iterator end() ;
 const_iterator end() const;
 reverse_iterator rbegin() ;
 const_reverse_iterator rbegin() const;
 reverse_iterator rend() ;
 const_reverse_iterator rend() const;
 // ...
};
```

- Ponieważ *string* ma wymagane typy składowe i funkcje do uzyskiwania iteratorów, więc można używać takich napisów w połączeniu ze standardowymi algorytmami

```
void f(string& s)
{
 string::iterator p = find(s.begin() ,s.end() ,´a´) ;
 // ...
}
```

- Najbardziej popularne operacje na napisach dostarcza bezpośrednio klasa string
- Nie kontroluje się zakresu iteratorów napisowych

Dostęp do elementu

- Do poszczególnych znaków napisu można się dostawać za pomocą indeksowania

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // element access
 const_reference operator[](size_type n) const; // unchecked access
 reference operator[](size_type n) ;
 const_reference at(size_type n) const; // checked access
 reference at(size_type n) ;
 // ...
};
```

- Przy próbie dostępu poza zakres *at()* zgłasza wyjątek *out_of_range*
- Brak funkcji *front()* i *back()*
- Równoważność wskaźników/tablic nie zachodzi dla napisów, *&s[0]* to nie to samo, co *s*

Konstruktory

- Zbiór operacji inicjowania i kopiowania różni się w wielu szczegółach od zbioru dostarczanego dla innych kolekcji

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // constructors, etc. (a bit like vector and list)
 explicit basic_string(const A& a =A()) ;
 basic_string(const basic_string& s,
 size_type pos = 0, size_type n = npos, const A& a =A()) ;
 basic_string(const Ch* p, size_type n, const A& a =A()) ;
 basic_string(const Ch* p, const A& a =A()) ;
 basic_string(size_type n,Ch c, const A& a =A()) ;
 template<class In> basic_string(In first, In last, const A& a =A()) ;
 ~basic_string() ;
 static const size_type npos; // ``all characters'' marker
 // ...
};
```

Konstruktory

- Napis typu *string* można zainicjować napisem w stylu C, innym napisem typu *string*, częścią napisu w stylu C, częścią *string* lub ciągiem znaków. Nie można go jednak zainicjować znakiem ani liczbą całkowitą

```
void f(char* p,vector<char>&v)
{
 string s0; // the empty string
 string s00= ""; // also the empty string
 string s1= ´a´; // error: no conversion from char to string
 string s2 = 7; // error: no conversion from int to string
 string s3(7) ; // error: no constructor taking one int argument
 string s4(7,´a´) ; // 7 copies of 'a'; that is "aaaaaaa"
 string s5= "Frodo"; // copy of "Frodo"
 string s6 = s5; // copy of s5
 string s7(s5,3,2) ; // s5[3] and s5[4]; that is "do"
 string s8(p+7,3) ; // p[7], p[8], and p[9]
 string s9(p,7,3) ; // string(string(p),7,3), possibly expensive
 string s10(v.begin() ,v.end()) ; // copy all characters from v
}
```

- Znaki są ponumerowane od pozycji 0, więc napis jest ciągiem znaków o numerach od 0 do *length()* -1
- *length()* jest synonimem *size()*, obie przekazują liczbę znaków w napisie
  - Nie liczą one zera kończącego napis w stylu C

Konstruktory

- Podnapisy wyraża się, podając pozycję znaku i liczbę znaków
- Konstruktor kopiujący przyjmuje cztery argumenty, trzy z nich mają wartości domyślne
- Najbardziej ogólny jest konstruktor będący składową wzorca. Umożliwia on inicjowanie napisu wartościami z dowolnego ciągu, a zwłaszcza elementami innego typu znakowego, jeśli tylko istnieje konwersja

```
void f(string s)
{
 wstring ws(s.begin() ,s.end()) ; // copy all characters from s
 // ...
}
```

Błędy

- Manipulacje na podnapisach lub znakach mogą być źródłem błędów - pisania za końcem napisu
- *at()* zgłasza *out_of_range* przy próbie dostępu za koniec napisu
- Większość operacji napisowych przyjmuje pozycję znaku i licznik znaków. Podanie pozycji większej niż rozmiar napisu powoduje zgłoszenie wyjątku *out_of_range*. "Zbyt duży" licznik znaków jest po prostu traktowany jako równoważny "reszcie znaków"

```
void f()
{
 string s= "Snobol4";
 string s2(s,100,2) ; // character position beyond end of string:
                      // throw out_of_range()
 string s3(s,2,100) ; // character count too large:
                      // equivalent to s3(s,2,s.size()- 2)
 string s4(s,2,string::npos) ; // the characters starting from s[2]
}
```

- Liczba ujemna jest mylącym sposobem podania dużej liczby dodatniej, bo typ *size_type* użyty do reprezentowania pozycji i liczników jest typem bez znaku

```
void g(string& s)
{
 string s5(s,-2,3); // large position!: throw out_of_range()
 string s6(s,3,-2); // large character count!: ok
}
```

- Funkcje używane do znajdowania podnapisów w napisie zwracają *npos*, gdy nic nie znajdą. Próba użycia *npos* jako pozycji znaku zgłosi wyjątek
- Dla wszystkich napisów zachodzi warunek *length()* < *npos*. Niekiedy, podczas wstawiania jednego napisu do drugiego, można skonstruować napis zbyt długi, by można go było reprezentować, wówczas zgłaszany jest *length_error*

Przypisanie

- Na napisach można wykonywać przypisanie

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // assignment (a bit like vector and list: §16.3.4):
 basic_string& operator=(const basic_string& s) ;
 basic_string& operator=(const Ch* p) ;
 basic_string& operator=(Ch c) ;
 basic_string& assign(const basic_string&) ;
 basic_string& assign(const basic_string& s, size_type pos, size_type n) ;
 basic_string& assign(const Ch* p, size_type n) ;
 basic_string& assign(const Ch* p) ;
 basic_string& assign(size_type n,Ch c) ;
 template<class In> basic_string& assign(In first, In last) ;
 // ...
};
```

Przypisanie

- *string* ma semantykę wartości
  - przypisując jeden napis na drugi, kopiuje się przypisywany napis, a po wykonaniu przypisania istnieją dwa odrębne napisy o tej samej wartości

```
void g()
{
 string s1= "Knold";
 string s2= "Tot";
 s1 = s2; // two copies of "Tot"
 s2[1] = ´u´; // s2 is "Tut", s1 is still "Tot"
}
```

- Przypisanie napisowi pojedynczego znaku jest dopuszczalne, mimo że nie jest tak podczas inicjowania

```
void f()
{
 string s= ´a´; // error: initialization by char
 s= ´a´; // ok: assignment
 s= "a";
 s = s;
}
```

- Nazwy *assign()* używa się na określenie przypisań, które są odpowiednikami jednoargumentowych konstruktorów

Konwersja do napisu w stylu C

- Można zainicjować *string* napisem w stylu C i można mu przypisać taki napis. Można też umieścić kopię znaków obiektu klasy *string* w tablicy

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // conversion to C-style string:
 const Ch* c_str() const;
 const Ch* data() const;
 size_type copy(Ch* p, size_type n, size_type pos = 0) const;
 // ...
};
```

- Funkcja *data()* wpisuje znaki napisu do tablicy i przekazuje wskaźnik do niej.
  - Tablica jest własnością obiektu klasy *string* i użytkownik nie powinien próbować jej usuwać
  - Nie powinien polegać na jej zawartości po kolejnym wywołaniu dla napisu funkcji bez atrybutu *const*

Konwersja do napisu w stylu C

- Funkcja *c_str()* jest podobna do *data()*, ale dodaje 0 na końcu jako terminator napisu w stylu C

```
void f()
{
 string s= "equinox"; // s.length()==7
 const char* p1 = s.data() ; // p1 points to seven characters
 printf("p1= %s\n",p1) ; // bad: missing terminator
 p1[2] = ´a´; // error: p1 points to a const array
 s[2] = ´a´;
 char c = p1[1] ; // bad: access of s.data() after modification of s
 const char* p2 = s.c_str() ; // p2 points to eight characters
 printf("p2= %s\n",p2) ; // ok: c_str() adds terminator
}
```

- Głównym zadaniem funkcji konwersji jest dopuszczenie prostego użycia funkcji o argumentach będących napisami w stylu C. *c_str()* jest tu bardziej użyteczna niż *data()*

```
void f(string s)
{
 int i = atoi(s.c_str()) ; // get int value of digits in string
 // ...
}
```

Konwersja do napisu w stylu C

- Zwykle najlepiej jest pozostawić znaki w napisie do czasu, aż będą potrzebne. Jeżeli nie można użyć ich natychmiast, można je skopiować do tablicy, zamiast pozostawiać w buforze przydzielonym przez *c_str()* lub *data()*

```
char* c_string(const string& s)
{
 char* p = new char[s.length()+1] ; // note: +1
 s.copy(p,string::npos) ;
 p[s.length()] = 0; // note: add terminator
 return p;
}
```

- Wywołanie *s.copy(p, n, m)* kopiuje co najwyżej *n* znaków do *p* począwszy od *s[m]*. Jeżeli w *s* do skopiowania jest mniej niż *n* znaków, to funkcja kopiuje wszystkie znaki, które tam są
- *string* może zawierać znak 0. Funkcje obsługujące napisy w stylu C zinterpretują go jako terminator.

Porównania

- Napisy można porównywać z napisami tego samego typu i tablicami znaków o tym samym typie znakowym

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 int compare(const basic_string& s) const; // combined > and ==
 int compare(const Ch* p) const;
 int compare(size_type pos, size_type n, const basic_string& s) const;
 int compare(size_type pos, size_type n,
                const basic_string& s, size_type pos2, size_type n2) const;
 int compare(size_type pos, size_type n, const Ch* p,
                                                size_type n2 = npos) const;
 // ...
};
```

- Gdy jest dostępny argument *n*, porównuje się tylko *n* pierwszych znaków. Używanym kryterium porównawczym jest *compare()* z *char_traits<Ch>*.
- Użytkownik nie może dostarczyć kryterium porównawczego

Porównania

- Dla obiektów klasy basic_string dostępne są zwykłe operatory porównania ==, !=, >,<,>=,<=

```
template<class Ch, class Tr, class A>
                              bool operator==(const basic_string<Ch,Tr,A>&,
                                              const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
                    bool operator==(const Ch*, const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
                    bool operator==(const basic_string<Ch,Tr,A>&, const Ch*) ;
   // similar declarations for !=, >, <, >=, and <=
```

- Operatory porównania nie są metodami, więc można stosować konwersje w ten sam sposób dla obu argumentów. Wersje przyjmujące argumenty w stylu C optymalizują porównania z literałami napisowymi

```
void f(const string& name)
{
 if (name =="Obelix" || "Asterix"==name) { // use optimized ==
 // ...
 }
}
```

Wstawianie

- Jedną z najbardziej popularnych operacji modyfikujących wartość napisu jest dołączanie do niego, tzn. dodawanie znaków na końcu. Wstawianie znaków w inne miejsca jest rzadsze

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // add characters after (*this)[length()- 1]:
 basic_string& operator+=(const basic_string& s) ;
 basic_string& operator+=(const Ch* p) ;
 basic_string& operator+=(Ch c) ;
 void push_back(Ch c) ;
 basic_string& append(const basic_string& s) ;
 basic_string& append(const basic_string& s, size_type pos, size_type n) ;
 basic_string& append(const Ch* p, size_type n) ;
 basic_string& append(const Ch* p) ;
 basic_string& append(size_type n,Ch c) ;
 template<class In> basic_string& append(In first, In last) ;
 // insert characters before (*this)[pos]:
 basic_string& insert(size_type pos, const basic_string& s) ;
 basic_string& insert(size_type pos, const basic_string& s, size_type pos2,
                                                            size_type n) ;
 basic_string& insert(size_type pos, const Ch* p, size_type n) ;
 basic_string& insert(size_type pos, const Ch* p) ;
 basic_string& insert(size_type pos, size_type n,Ch c) ;
 // insert characters before p:
 iterator insert(iterator p,Ch c) ;
 void insert(iterator p, size_type n,Ch c) ;
 template<class In> void insert(iterator p, In first, In last) ;
 // ...
};
```

Wstawianie

- Zasadniczo zestaw różnych operacji dostarczonych do inicjowania napisu i przypisywania mu wartości jest także dostępny do dołączania i wstawiania znaków przed pewną pozycją
- Operator += pełni funkcję notacji opisującej najbardziej popularne postacie dołączania

```
string complete_name(const string& first_name, const string& family_name)
{
 string s = first_name;
 s += ´ ´;
 s += family_name;
 return s;
}
```

- Dołączanie na koniec może być zauważalnie bardziej efektywne niż wstawianie na inne pozycje

```
string complete_name2(const string& first_name, const string& family_name)
                                                        // poor algorithm
{
 string s = family_name;
 s.insert(s.begin() ,´ ´) ;
 return s.insert(0,first_name) ;
}
```

Konkatenacja

- Dołączanie jest szczególną postacią konkatenacji. Konkatenacja - konstruowanie napisu z dwóch innych przez umieszczenie jednego za drugim - jest dostępna w postaci operatora +

```
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A>
 operator+(const basic_string<Ch,Tr,A>&, const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A> operator+(const Ch*, const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A> operator+(Ch, const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&, const Ch*) ;
template<class Ch, class Tr, class A>
 basic_string<Ch,Tr,A> operator+(const basic_string<Ch,Tr,A>&,Ch) ;
```

- Jak zwykle, + jest zdefiniowana jako funkcja nie będąca metodą
- Użycie konkatenacji jest oczywiste i wygodne

```
string complete_name3(const string& first_name, const string& family_name)
{
 return first_name+ ´ ´ + family_name;
}
```

- Tę wygodę notacyjną uzyskuje się kosztem pewnego narzutu w czasie wykonania w porównaniu z complete_name(). Funkcja complete_name3() potrzebuje jednej dodatkowej zmiennej tymczasowej

Wyszukiwanie

- Istnieje dużo różnych funkcji do wyszukiwania napisów

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
class basic_string {
 public:
 // ...
 // find subsequence (like search()):
 size_type find(const basic_string& s, size_type i = 0) const;
 size_type find(const Ch* p, size_type i, size_type n) const;
 size_type find(const Ch* p, size_type i = 0) const;
 size_type find(Ch c, size_type i = 0) const;
 // find subsequence searching backwards from the end (like find_end()):
 size_type rfind(const basic_string& s, size_type i = npos) const;
 size_type rfind(const Ch* p, size_type i, size_type n) const;
 size_type rfind(const Ch* p, size_type i = npos) const;
 size_type rfind(Ch c, size_type i = npos) const;
 // find character (like find_first_of()):
 size_type find_first_of(const basic_string& s, size_type i = 0) const;
 size_type find_first_of(const Ch* p, size_type i, size_type n) const;
 size_type find_first_of(const Ch* p, size_type i = 0) const;
 size_type find_first_of(Ch c, size_type i = 0) const;
 // find character from argument searching backwards from the end:
 size_type find_last_of(const basic_string& s, size_type i = npos) const;
 size_type find_last_of(const Ch* p, size_type i, size_type n) const;
 size_type find_last_of(const Ch* p, size_type i = npos) const;
 size_type find_last_of(Ch c, size_type i = npos) const;
```

Wyszukiwanie

```
  // find character not in argument:
  size_type find_first_not_of(const basic_string& s, size_type i = 0) const;
  size_type find_first_not_of(const Ch* p, size_type i, size_type n) const;
  size_type find_first_not_of(const Ch* p, size_type i = 0) const;
  size_type find_first_not_of(Ch c, size_type i = 0) const;
  // find character not in argument searching backwards from the end:
  size_type find_last_not_of(const basic_string& s, size_type i=npos) const;
  size_type find_last_not_of(const Ch* p, size_type i, size_type n) const;
  size_type find_last_not_of(const Ch* p, size_type i = npos) const;
  size_type find_last_not_of(Ch c, size_type i = npos) const;
  // ...
};
```

- Są to wszystko metody z atrybutem *const*, tzn. pozwalają zlokalizować podnapis w jakimś celu, lecz nie zmieniają jego wartości
- Znaczenie funkcji *basic_string::find()* można zrozumieć na podstawie równoważnych im ogólnych algorytmów

```
void f() {
  string s= "accdcde";
  string::size_type i1 = s.find("cd") ; // i1 = 2 s[2]=='c' && s[3]=='d'
  string::size_type i2 = s.rfind("cd") ; // i2 = 4 s[4]=='c' && s[5]=='d'
  string::size_type i3 = s.find_first_of("cd") ; // i3 = 1 s[1] == 'c'
  string::size_type i4 = s.find_last_of("cd") ; // i4 = 5 s[5] == 'd'
  string::size_type i5 = s.find_first_not_of("cd") ;
                                          // i5 = 0 s[0]!='c' && s[0]!='d'
  string::size_type i6 = s.find_last_not_of("cd") ;
                                         // i6 = 6 s[6]!='c' && s[6]!='d'
}
```

- Jeżeli funkcja *find()* nic nie znajdzie, to przekazuje *npos* reprezentujące nielegalną pozycję znaku. Jeżeli użyje się *npos* do wskazania pozycji, to zostanie zgłoszony wyjątek *out_of_range*

Zastępowanie

- Po zidentyfikowaniu pozycji w napisie można zmienić wartość pojedynczej pozycji za pomocą indeksowania lub zastąpić cały podnapis nowymi znakami. Służy do tego funkcja *replace()*

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // replace [ (*this)[i], (*this)[i+n] [ with other characters:
 basic_string& replace(size_type i, size_type n, const basic_string& s) ;
 basic_string& replace(size_type i, size_type n,
 const basic_string& s, size_type i2, size_type n2) ;
 basic_string& replace(size_type i, size_type n, const Ch* p, size_type n2) ;
 basic_string& replace(size_type i, size_type n, const Ch* p) ;
 basic_string& replace(size_type i, size_type n, size_type n2,Ch c) ;
 basic_string& replace(iterator i, iterator i2, const basic_string& s) ;
 basic_string& replace(iterator i, iterator i2, const Ch* p, size_type n) ;
 basic_string& replace(iterator i, iterator i2, const Ch* p) ;
 basic_string& replace(iterator i, iterator i2, size_type n,Ch c) ;
 template<class In>basic_string& replace(iterator i,iterator i2, In j,In j2);
 // remove characters from string (``replace with nothing''):
 basic_string& erase(size_type i = 0, size_type n = npos) ;
 iterator erase(iterator i) ;
 iterator erase(iterator first, iterator last) ;
 // ...
};
```

Zastępowanie

- Liczba nowych znaków nie musi być taka sama jak liczba znaków poprzednio występujących w napisie. Rozmiar napisu zmienia się, dostosowując się do nowego podnapisu.
- Funkcja *erase()* po prostu usuwa podnapis i odpowiednio poprawia swój rozmiar

```
void f()
{
 string s= "but I have heard it works even if you don´t believe in it";
 s.erase(0,4) ; // erase initial "but "
 s.replace(s.find("even") ,4,"only") ;
 s.replace(s.find("don´t") ,5,"") ; // erase by replacing with ""
}
```

- Proste wywołanie *erase()* bez argumentów zamienia napis w napis pusty. Jest to operacja, która w odniesieniu do ogólnych kolekcji nazywa się *clear()*
- Różnorodność funkcji *replace()* odpowiada różnorodności przypisań; *replace()* jest przecież przypisaniem na podnapis

Podnapisy

- Funkcja *substr()* pozwala wyspecyfikować podnapis przez podanie pozycji i długości

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // address substring:
 basic_string substr(size_type i = 0, size_type n = npos) const;
 // ...
};
```

Rozmiar i pojemność

- Kwestie związane z zarządzaniem pamięcią obsługuje się z grubsza tak, jak dla klasy wektor

```
template<class Ch, class Tr = char_traits<Ch>, class A = allocator<Ch> >
 class basic_string {
 public:
 // ...
 // size, capacity, etc.
 size_type size() const; // number of characters
 size_type max_size() const; // largest possible string
 size_type length() const { return size() ; }
 bool empty() const { return size()==0; }
 void resize(size_type n,Ch c) ;
 void resize(size_type n) { resize(n,Ch()) ; }
 size_type capacity() const; // like vector
 void reserve(size_type res_arg = 0) ; // like vector
 allocator_type get_allocator() const;
};
```

- Wywołanie funkcji *reserve(res_arg)* powoduje zgłoszenie wyjątku *length_error*, gdy *res_arg>max_size()*

Operacje wejścia-wyjścia

- Napisy są często stosowane jako wynik wejścia i źródło wyjścia

```
template<class Ch, class Tr, class A>
 basic_istream<Ch,Tr>& operator>>(basic_istream<Ch,Tr>&,
                                  basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_ostream<Ch,Tr>& operator<<(basic_ostream<Ch,Tr>&,
                            const basic_string<Ch,Tr,A>&) ;
template<class Ch, class Tr, class A>
 basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&,
                         basic_string<Ch,Tr,A>&,Ch eol) ;
template<class Ch, class Tr, class A>
 basic_istream<Ch,Tr>& getline(basic_istream<Ch,Tr>&,
                                  basic_string<Ch,Tr,A>&) ;
```

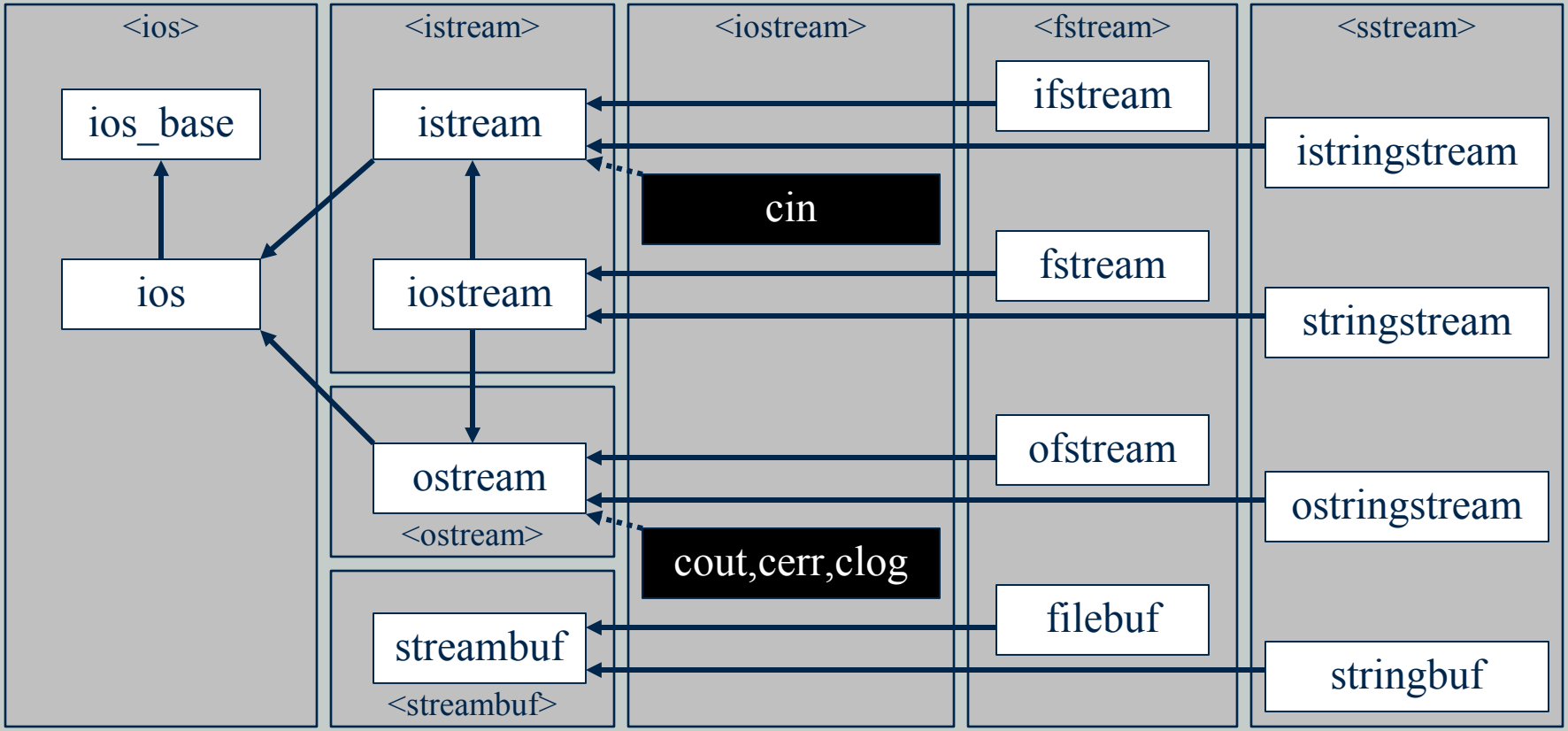
- Operator << wypisuje napis na *ostream*, a >> czyta słowo zakończone białym znakiem do napisu, odpowiednio go rozszerzając. Początkowe białe znaki są omijane, a końcowy biały znak nie jest wprowadzany do napisu
- Funkcja *getline()* czyta do napisu wiersz zakończony znakiem *eol*, odpowiednio go rozszerzając. Jeśli nie dostarczono argumentu *eol*, to jako ogranicznika używa się znaku przejścia do nowego wiersza *'\n'*. Terminator wiersza usuwa się ze strumienia, ale nie wprowadza się go do napisu. Ponieważ napis rozszerza się, by zmieścić wejście, nie ma powodu, by zostawić terminator w strumieniu lub dostarczać licznik wczytanych znaków, tak jak to robi *get()* i *getline()* dla tablic znakowych.

Zamiana

- Podobnie jak dla wektorów, funkcja *swap()* dla napisów może być dużo bardziej efektywna, niż ogólny algorytm, więc dostarcza się specjalną wersję

```
template<class Ch, class Tr, class A>
 void swap(basic_string<Ch,Tr,A>&, basic_string<Ch,Tr,A>&) ;
```

Struktura klas



Wyjście

- Można osiągnąć bezpieczne i jednolite traktowanie zarówno typów wbudowanych, jak i zdefiniowanych przez użytkownika, używając pojedynczej przeciążonej nazwy funkcji dla zbioru funkcji wyjścia

```
put(cerr,"x= ") ; // cerr is the error output stream
put(cerr,x) ;
put(cerr,´\n´) ;
```

- Bardziej zwięzłą formą operacji wyprowadzania jest przeciążenie operatora <<, co daje bardziej zwięzłą notację i pozwala programiście na umieszczenie ciągu obiektów w pojedynczej instrukcji

```
cerr << "x= " << x << ´\n´;
```

- Priorytet << jest na tyle niski, że można używać wyrażeń arytmetycznych jako operandów bez stosowania nawiasów

```
cout << "a*b+c=" << a*b+c << ´\n´;
```

- Nawiasy trzeba stosować do zapisywania wyrażeń zawierających operatory o niższym priorytecie

```
cout << "a^b|c=" << (a^b|c) << ´\n´;
```

- Operator << może być użyty w instrukcji wyjścia jako operator przesunięcia w lewo, ale wówczas należy go umieścić w nawiasach

```
cout << "a<<b=" << (a<<b) << ´\n´;
```

Strumienie wyjściowe

- Klasa *ostream* jest mechanizmem do konwersji wartości różnych typów na ciągi znaków
- *ostream* jest specjalizacją ogólnego wzorca *basic_ostream*

```
template <class Ch, class Tr = char_traits<Ch> >
 class std::basic_ostream : virtual public basic_ios<Ch,Tr> {
 public:
 virtual ~basic_ostream() ;
 // ...
};
```

- Każda implementacja bezpośrednio wspiera strumienie realizowane za pomocą zwykłych i szerokich znaków

```
typedef basic_ostream<char> ostream;
typedef basic_ostream<wchar_t>wostream;
```

Strumienie wyjściowe

- Klasa podstawowa *basic_ios* umożliwia kontrolę formatowania, lokalizmów i dostępu do buforów. Zawiera także definicje kilku typów upraszczających notację

```
template <class Ch, class Tr = char_traits<Ch> >
 class std::basic_ios : public ios_base {
 public:
 typedef Ch char_type;
 typedef Tr traits_type;
 typedef typename Tr::int_type int_type; // type of integer value
                                         // of character
 typedef typename Tr::pos_type pos_type; // position in buffer
 typedef typename Tr::off_type off_type; // offset in buffer
 //...
};
```

- *basic_ios* nie dopuszcza konstruktora kopiującego ani przypisania - strumieni nie można kopiować
- Klasa *ios_base* zawiera informacje i operacje niezależne od użytego typu znaku - nie musi być wzorcem

Strumienie wyjściowe

- Biblioteka strumieniowa wejścia-wyjścia, oprócz definicji typów w ios_base, używa typu całkowitego ze znakiem, *streamsize*, do reprezentowania liczby znaków przesłanych w operacji wejścia-wyjścia i rozmiaru buforów
- *streamoff* służy do wyrażania przesunięcia w strumieniach i buforach
- W <*iostream*> zadeklarowano kilka standardowych strumieni
  - *cerr* i *clog* odpowiadają *stderr*
  - *cout* odpowiada *stdout*

```
ostream cout; // standard output stream of char
ostream cerr; // standard unbuffered output stream for error messages
ostream clog; // standard output stream for error messages
wostream wcout; // wide stream corresponding to cout
wostream wcerr; // wide stream corresponding to cerr
wostream wclog; // wide stream corresponding to clog
```

Strumienie wejściowe

- W *<istream>* istnieje *basic_istream*

```
template <class Ch, class Tr = char_traits<Ch> >
 class std::basic_istream : virtual public basic_ios<Ch,Tr> {
 public:
 virtual ~basic_istream() ;
 // ...
};
```

- Nagłówek *<iostream>* dostarcza dwa standardowe strumienie wejściowe

```
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t>wistream;
istream cin; // standard input stream of char
wistream wcin; // standard input stream of wchar_t
```

Stan strumienia

- Z każdym strumieniem związany jest jego stan. Właściwe ustawianie i testowanie stanu strumienia umożliwia obsługę błędów i warunków niestandardowych

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ios : public ios_base {
 public:
 // ...
 bool good() const; // next operation might succeed
 bool eof() const; // end of input seen
 bool fail() const; // next operation will fail
 bool bad() const; // stream is corrupted
 iostate rdstate() const; // get io state flags
 void clear(iostate f = goodbit) ; // set io state flags
 void setstate(iostate f) {clear(rdstate()|f) ;} // add f to io state flags
 operator void*() const; // nonzero if !fail()
 bool operator!() const { return fail() ; }
 // ...
};
```

- Jeżeli stan jest *good()*, to poprzednia operacja się powiodła, a następna może się powieść, w przeciwnym razie zakończy się niepowodzeniem
- Jeżeli próba czytania na zmienną *v* się nie powiedzie, wartość *v* nie powinna się zmienić (tak jest w przypadku typów standardowo obsługiwanych przez *istream*)
- Różnica między stanami *fail()* i *bad()* jest bardzo subtelna
  - w stanie *fail()* zakłada się, że strumień nie jest zniszczony i nie zgubiono żadnych znaków
  - w stanie *bad()* nie ma takiej gwarancji

Stan strumienia

- Stan strumienia jest reprezentowany przez zbiór znaczników

```
class ios_base {
 public:
 // ...
 typedef implementation_defined2 iostate;
 static const iostate badbit, // stream is corrupted
 eofbit, // end-of-file seen
 failbit, // next operation will fail
 goodbit; // goodbit==0
 // ...
};
```

- Znacznikami stanu wejścia-wyjścia można bezpośrednio manipulować

```
void f()
{
 ios_base::iostate s = cin.rdstate() ; // returns a set of iostate bits
 if (s & ios_base::badbit) {
   // cin characters possibly lost
 }
 // ...
 cin.setstate(ios_base::failbit) ;
 // ...
}
```

Stan strumienia

- Gdy używa się strumienia zamiast warunku, stan strumienia testuje się za pomocą operator *void*()* lub *operator!()*. Tet kończy się sukcesem tylko wtedy, gdy stanem jest odpowiednio *!fail()* i *fail()*
- Ogólną funkcję kopiującą można zapisać następująco

```
template<class T> void iocopy(istream& is, ostream& os)
{
 T buf;
 while (is>>buf) os << buf << ´\n´;
}
```

- Operacja *is>>buf* przekazuje referencję do strumienia *is*, testowanego wywołaniem *is::operator void*()*

```
void f(istream& i1, istream& i2, istream& i3, istream& i4)
{
 iocopy<complex>(i1,cout) ; // copy complex numbers
 iocopy<double>(i2,cout) ; // copy doubles
 iocopy<char>(i3,cout) ; // copy chars
 iocopy<string>(i4,cout) ; // copy whitespace-separated words
}
```

Wejście niesformatowane

- Operator >> służy do formatowanego wejścia, czyli czytania obiektów oczekiwanego typu i formatu. Jeśli nie to jest naszym celem, bo chcemy czytać znaki jako znaki, a potem je analizować, używamy funkcji *get()*

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_istream : virtual public basic_ios<Ch,Tr> {
 public:
 // ...
 // unformatted input:
 streamsize gcount() const; // number of char read by last get()
 int_type get() ; // read one Ch (or Tr::eof())
 basic_istream& get(Ch& c) ; // read one Ch into c
 basic_istream& get(Ch* p, streamsize n) ;
           // newline is terminator, doesn't remove terminator from stream
 basic_istream& get(Ch* p, streamsize n, Ch term) ;
                      // reads at most n-1 characters, terminates with \0
 basic_istream& getline(Ch* p, streamsize n) ;
                  // newline is terminator, removes terminator from stream
 basic_istream& getline(Ch* p, streamsize n, Ch term) ;
 basic_istream& ignore(streamsize n = 1, int_type t = Tr::eof()) ;
 basic_istream& read(Ch* p, streamsize n) ; // read at most n char
 // ...
};
```

Wyjątki

- Ponieważ sprawdzanie poprawności każdej operacji wejścia-wyjścia jest niewygodne, więc częstą przyczyną błędu jest zaniedbywanie tego przez programistę
- Jedyną funkcją bezpośrednio zmieniającą stan strumienia jest *clear()*. Za pomocą funkcji *exceptions()* można nakazać tej funkcji zgłaszanie wyjątków

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ios : public ios_base {
 public:
 // ...
 class failure; // exception class (see §14.10)
 iostate exceptions() const; // get exception state
 void exceptions(iostate except) ; // set exception state
 // ...
};
```

- Wywołanie

```
cout.exceptions(ios_base::badbit|ios_base::failbit|ios_base::eofbit) ;
```

jest żądaniem zgłoszenia przez *clear()* wyjątku *ios_base::failure()* gdy *cout* przejdzie w stan *bad*, *fail* lub *eof*

- Wywołanie *exceptions()* bez argumentów zwraca zbiór znaczników wejścia-wyjścia, które mogą spowodować wyjątek

Formatowanie

- Formatowaniem wejścia-wyjścia steruje zbiór znaczników i wartości całkowitych z klasy *ios_base*

```
class ios_base {
 public:
 // ...
 // names of format flags:
 typedef implementation_defined1 fmtflags;
 static const fmtflags
   skipws, // skip whitespace on input
   left, // field adjustment: pad after value
   right, // pad before value
   internal, // pad between sign and value
   boolalpha, // use symbolic representation of true and false
   dec, // integer base: base 10 output (decimal)
   hex, // base 16 output (hexadecimal)
   oct, // base 8 output (octal)
   scientific, // floating-point notation: d.ddddddEdd
   fixed, // dddd.dd
   showbase, // on output prefix oct by 0 and hex by 0x
   showpoint, // print trailing zeros
   showpos, // explicit '+' for positive ints
   uppercase, // 'E', 'X' rather than 'e', 'x'
   adjustfield, // flags related to field adjustment
   basefield, // flags related to integer base
   floatfield, // flags related to floating-point output
   unitbuf; // flush output after each output operation
 fmtflags flags() const; // read flags
 fmtflags flags(fmtflags f) ; // set flags
 fmtflags setf(fmtflags f) { return flags(flags()|f) ; } // add flag
 fmtflags setf(fmtflags f, fmtflags mask)
                    { return flags(flags()|(f&mask)) ; } // add flag
 void unsetf(fmtflags mask) { flags(flags()&~mask) ; } // clear flags
 // ...
};
```

Formatowanie

- Definiowanie interfejsu jako zbioru znaczników i operacji do ustawiania oraz czyszczenia znaczników jest dość staromodną techniką. Jej główną zaletą jest możliwość układania zbioru opcji przez użytkownika

```
const ios_base::fmtflags my_opt =
                                ios_base::left|ios_base::oct|ios_base::fixed;
```

- Taki zbiór opcji można przekazywać i instalować w miarę potrzeby

```
void your_function(ios_base::fmtflags opt)
{
 ios_base::fmtflags old_options = cout.flags(opt) ;
                                        // save old_options and set new ones
 // ...
 cout.flags(old_options) ; // reset options
}
void my_function()
{
 your_function(my_opt) ;
 // ...
}
```

- Funkcja *flags()* przekazuje stary zbiór opcji

Formatowanie

- Możliwość czytania i ustawiania wszystkich opcji pozwala także ustawić pojedynczy znacznik.
- Instrukcja

```
myostream.flags(myostream.flags()|ios_base::showpos) ;
```

  spowoduje wyświetlenie jawnego znaku + przed każdą liczbą dodatnią umieszczoną w strumieniu *myostream*, nie wpływając na pozostałe opcje. Najpierw odczytuje się stare opcje, a następnie do uzyskanego wzorca bitowego dodaje się bit *showpos*.
- Funkcja *setf()* robi dokładnie to samo, więc można przykład zapisać równoznacznie w następujący sposób

```
myostream.setf(ios_base::showpos) ;
```

- Znacznik po ustawieniu zachowuje swoją wartość aż do jawnego wyzerowania

Kopiowanie stanu strumienia

- Można skopiować pełny stan formatowania strumienia z pomocy *copyfmt()*

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ios : public ios_base {
 public:
 // ...
 basic_ios& copyfmt(const basic_ios& f) ;
 // ...
};
```

- Funkcja nie kopiuje bufora strumienia ani jego stanu. Kopiuje natomiast cały stan, włącznie z żądanymi wyjątkami i dowolnymi dostarczonymi przez użytkownika rozszerzeniami tego stanu

Wyjście dla liczb całkowitych

- Technika dodawania (operacją alternatywy bitowej) nowej opcji za pomocą *flags()* lub *setf()* działa wtedy, kiedy daną cechę kontroluje pojedynczy bit
- Nie jest tak dla opcji określających podstawę używaną do drukowania liczb całkowitych i określających styl wyjścia zmiennopozycyjnego
  - Wartości specyfikującej styl nie można reprezentować pojedynczym bitem czy zbiorem niezależnych pojedynczych bitów
- Rozwiązanie przyjęte w *<iostream>* polega na dostarczeniu wersji funkcji *setf()*, która, oprócz nowej wartości, przyjmuje drugi argument wskazujący, jakiego rodzaju opcję chcemy ustawić

Wyjście dla liczb całkowitych

- Instrukcje

```
cout.setf(ios_base::oct,ios_base::basefield) ; // octal
cout.setf(ios_base::dec,ios_base::basefield) ; // decimal
cout.setf(ios_base::hex,ios_base::basefield) ; // hexadecimal
```

ustawiają podstawę arytmetyki całkowitej, bez ubocznego wpływu na stan strumienia. Obowiązuje ona do następnej zmiany

- Instrukcje

```
cout << 1234 << ´ ´ << 1234 << ´ ´; //default: decimal
cout.setf(ios_base::oct,ios_base::basefield) ; // octal
cout << 1234 << ´ ´ << 1234 << ´ ´;
cout.setf(ios_base::hex,ios_base::basefield) ; // hexadecimal
cout << 1234 << ´ ´ << 1234 << ´ ´;
```

dają w wyniku

```
1234 1234 2322 2322 4d2 4d2
```

- Jeśli chcemy wiedzieć, jakiej podstawy użyto dla każdej liczby, możemy ustawić *showbase*. Jeśli dodamy przed podanymi instrukcjami

```
cout.setf(ios_base::showbase) ;
```

to otrzymamy

```
1234 1234 02322 02322 0x4d2 0x4d2
```

Wyjście dla liczb zmiennopozycyjnych

- O postaci wyjścia dla liczb zmiennopozycyjnych decyduje format i precyzja
  - format ogólny umożliwia implementacji wybór formatu prezentującego wartość w stylu, który najlepiej przedstawia wartość w dostępnej przestrzeni. Precyzja określa maksymalną liczbę cyfr. Odpowiada opcji %g funkcji *printf()*
  - Format naukowy prezentuje wartość z jedną cyfrą przed kropką dziesiętną i wykładnikiem. Precyzja określa maksymalną liczbę cyfr po kropce. Odpowiada opcji %e funkcji *printf()*
  - Format stały prezentuje wartość jako część całkowitą, po której następuje kropka dziesiętna i część ułamkowa. Precyzja określa maksymalną liczbę cyfr po kropce. Odpowiada opcji %f funkcji *printf()*
- Formatem wyjścia zmiennopozycyjnego steruje się za pomocą funkcji do modyfikacji stanu strumienia. W szczególności można ustalać notację stosowaną przy wypisywaniu wartości zmiennoprzecinkowych bez ubocznego wpływu na inne części strumienia

Wyjście dla liczb zmiennopozycyjnych

- Instrukcje

```
cout << "default:\t" << 1234.56789 << ´\n´;
cout.setf(ios_base::scientific,ios_base::floatfield) ; // use scientific format
cout << "scientific:\t" << 1234.56789 << ´\n´;
cout.setf(ios_base::fixed,ios_base::floatfield) ; // use fixedpoint format
cout << "fixed:\t" << 1234.56789 << ´\n´;
cout.setf(0,ios_base::floatfield) ; // reset to default (that is, general format)
cout << "default:\t" << 1234.56789 << ´\n´;
```

produkują

```
default: 1234.57
scientific: 1.234568e+03
fixed: 1234.567890
default: 1234.57
```

- Domyślną precyzją (wszystkich formatów) jest 6, kontroluje ją metoda klasy *ios_base*

```
class ios_base {
 public:
 // ...
 streamsize precision() const; // get precision
 streamsize precision(streamsize n) ; // set precision (and get old precision)
 // ...
};
```

Wyjście dla liczb zmiennopozycyjnych

- Wywołanie *precision()* wpływa na wszystkie operacje wejścia-wyjścia na liczbach zmiennopozycyjnych, aż do następnego wywołania *precision()*
- Wywołania

```
cout.precision(8) ;
cout << 1234.56789 << ´ ´ << 1234.56789 << ´ ´ << 123456 << ´\n´;
cout.precision(4) ;
cout << 1234.56789 << ´ ´ << 1234.56789 << ´ ´ << 123456 << ´\n´;
```

produkują

```
1234.5679 1234.5679 123456
1235 1235 123456
```

- Wartości są zaokrąglane, a nie obcinane, *precision()* nie wpływa na sposób wypisywania liczb całkowitych
- Znacznik *uppercase* decyduje o tym, czy w formacie naukowym do wskazania wykładnika użyje się e, czy E

Pola wyjściowe

- Często chcemy wypełnić tekstem określony obszar w wierszu wyjściowym
  - chcemy użyć n znaków i nie mniej
- W tym celu specyfikuje się szerokość pola i znak, którego należy użyć, gdy pole wymaga dopełnienia

```
class ios_base {
 public:
 // ...
 streamsize width() const; // get field width
 streamsize width(streamsize wide) ; // set field width
 // ...
};
template <class Ch, class Tr = char_traits<Ch> >
class basic_ios : public ios_base {
 public:
 // ...
 Ch fill() const; // get filler character
 Ch fill(Ch ch) ; // set filler character
 // ...
};
```

Pola wyjściowe

- Funkcja width() określa minimalną liczbę znaków, jaka będzie użyta podczas następnej operacji << z biblioteki standardowej, powodującej wypisanie wartości numerycznej, *bool*, napisu w stylu C, wskaźnika, wartości typu *string* i *bitset*

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

wypisze liczbę 12 poprzedzoną dwoma spacjami

- Znak wypełniający można zdefiniować za pomocą funkcji *fill()*

```
cout.width(4) ;
cout.fill(´#´) ;
cout << "ab"; // ##ab
```

- Domyślnym znakiem wypełniającym jest spacja, a domyślnym rozmiarem pola 0, oznaczające "tyle znaków, ile potrzeba". Domyślny rozmiar pola można przywrócić następująco

```
cout.width(0) ; // ``as many characters as needed''
```

Pola wyjściowe

- Funkcja *width(n)* ustawia minimalną liczbę znaków na *n*. Jeżeli dostarczy się ich więcej, to będą wydrukowane wszystkie. Instrukcje

```
cout.width(4) ;
cout << "abcdef";
```

dadzą w wyniku abcdef a nie jedynie abcd

- Wywołanie *width(n)* wpływa tylko na następną w kolejności operację wejścia-wyjścia, a więc instrukcje

```
cout.width(4) ;
cout.fill(´#´) ;
cout << 12 << ´:´ << 13;
```

wyprodukują ##12:13 a nie ##12###:##13

Wyrównywanie pola

- Można sterować ustawieniem znaków w polu, wywołując *setf()*

```
cout.setf(ios_base::left,ios_base::adjustfield) ; // left
cout.setf(ios_base::right,ios_base::adjustfield) ; // right
cout.setf(ios_base::internal,ios_base::adjustfield) ; // internal
```

- Podane instrukcje określają sposób wyrównywania wyjścia w ramach pola wyjściowego, którego rozmiar zdefiniowano za pomocą *ios_base::width()*, bez ubocznego wpływu na stan strumienia

```
cout.fill(´#´) ;
cout << ´(´;
cout.width(4) ;
cout << -12 << ") ,(";
cout.width(4) ;
cout.setf(ios_base::left,ios_base::adjustfield) ;
cout << -12 << ") ,(";
cout.width(4) ;
cout.setf(ios_base::internal,ios_base::adjustfield) ;
cout << - 12 << ")";
```

- Podane instrukcje wypiszą (#-12), (-12#), (-#12)
- Domyślnie obowiązuje wyrównywanie w prawo

Manipulatory

- Żeby uchronić programistę przed koniecznością obsługi stanu strumienia za pomocą znaczników, biblioteka standardowa dostarcza zbiór funkcji do manipulowania tym stanem
- Pomysł polega na tym, żeby wstawić operację, która modyfikuje stan między czytaniem lub pisaniem obiektów
- Można na przykład jawnie zażądać opróżnienia bufora wyjściowego

```
cout << x << flush << y << flush;
```

- W odpowiednim momencie wywołana zostanie funkcja *cout.flush()*

Manipulatory

- Robi to wersja operatora <<, która przyjmuje wskaźnik do obiektu funkcyjnego i wywołuje go

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ostream : virtual public basic_ios<Ch,Tr> {
 public:
 // ...
 basic_ostream& operator<<(basic_ostream& (*f)(basic_ostream&))
   { return f(*this) ; }
 basic_ostream& operator<<(ios_base& (*f)(ios_base&)) ;
 basic_ostream& operator<<(basic_ios<Ch,Tr>& (*f)(basic_ios<Ch,Tr>&)) ;
 // ...
};
```

- Żeby to działało, funkcja musi być nie-metodą lub statyczną metodą odpowiedniego typu
- *flush()* jest zdefiniowana następująco

```
template <class Ch, class Tr = char_traits<Ch> >
basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>& s)
{
 return s.flush() ; // call ostream's member flush()
}
```

Manipulatory

- Powyższe definicje zapewniają, że instrukcje

```
cout << flush;
```

są interpretowane jako

```
cout.operator<<(flush) ;
```

co powoduje wywołanie

```
flush(cout) ;
```

a to z kolei wywołuje

```
cout.flush() ;
```

- Wszystko to razem pozwala na wywołanie *basic_ostream::flush()* za pomocą notacji *cout<<flush*

Manipulatory

- Istnieje wiele operacji, które dobrze byłoby wykonać tuż przed lub tuż po operacji wejścia lub wyjścia, np.:

```
cout << x;
cout.flush() ;
cout << y;
cin.noskipws() ; // don't skip whitespace
cin >> x;
```

- Kiedy zapisuje się je jako osobne instrukcje, to logiczne powiązanie między operacjami nie jest oczywiste; a kiedy straci się logiczne powiązanie, to kod staje się mniej czytelny
- Pojęcie manipulatorów pozwala na bezpośrednie wstawienie takich operacji jak flush() i noskipws() do listy operacji wejścia lub wyjścia, np.:

```
cout << x << flush << y << flush;
cin >> noskipws >> x;
```

- Klasa *basic_istream* zawiera operatory >> do wywoływania manipulatorów w sposób podobny do klasy *basic_ostream*

Manipulatory z argumentami

- Manipulatory z argumentami też mogą być użyteczne. Można na przykład napisać

```
cout << setprecision(4) << angle;
```

  żeby wydrukować wartość zmiennopozycyjną *angle* z czterema cyframi

- Trzeba w tym celu spowodować, aby *setprecision* przekazało obiekt inicjowany na 4, który można wywołać i który wówczas sam wywołuje *cout.precision(4)*
- Taki manipulator jest obiektem funkcyjnym, wywoływanym nie przez (), a przez <<. Typ obiektu zależy od implementacji, lecz może mieć następującą postać

```
struct smanip {
 ios_base& (*f)(ios_base&,int) ; // function to be called
 int i;
 smanip(ios_base& (*ff)(ios_base&,int) , int ii) : f(ff) , i(ii) { }
};
template<cladd Ch, class Tr>
ostream<Ch,Tr>& operator<<(ostream<Ch,Tr>& os, smanip&m)
{
 return m.f(os,m.i) ;
}
```

- Konstruktor klasy *smanip* zapamiętuje swoje argumenty w *f* oraz *i*, a *operator<<* wywołuje *f(i)*

Manipulatory z argumentami

- Funkcję *setprecision()* można zdefiniować następująco

```
ios_base& set_precision(ios_base& s, int n) // helper
{
 return s.setprecision(n) ; // call the member function
}
inline smanip setprecision(int n)
{
 return smanip(set_precision,n) ; // make the function object
}
```

- Teraz można napisać

```
cout << setprecision(4) << angle;
```

Standardowe manipulatory

```
ios_base& boolalpha(ios_base&) ; // symbolic representation of true and false
                                 // (input and output)
ios_base& noboolalpha(ios_base& s) ; // s.unsetf(ios_base::boolalpha)
ios_base& showbase(ios_base&) ; // on output prefix oct by 0 and hex by 0x
ios_base& noshowbase(ios_base& s) ; // s.unsetf(ios_base::showbase)
ios_base& showpoint(ios_base&) ;
ios_base& noshowpoint(ios_base& s) ; // s.unsetf(ios_base::showpoint)
ios_base& showpos(ios_base&) ;
ios_base& noshowpos(ios_base& s) ; // s.unsetf(ios_base::showpos)
ios_base& skipws(ios_base&) ; // skip whitespace
ios_base& noskipws(ios_base& s) ; // s.unsetf(ios_base::skipws)
ios_base& uppercase(ios_base&) ; // X and E rather than x and e
ios_base& nouppercase(ios_base&) ; // x and e rather than X and E
ios_base& internal(ios_base&) ; // adjust
ios_base& left(ios_base&) ; // pad after value
ios_base& right(ios_base&) ; // pad before value
ios_base& dec(ios_base&) ; // integer base is 10
ios_base& hex(ios_base&) ; // integer base is 16
ios_base& oct(ios_base&) ; // integer base is 8
ios_base& fixed(ios_base&) ; // floatingpoint format dddd.dd
ios_base& scientific(ios_base&) ; // scientific format d.ddddEdd
template <class Ch, class Tr>
 basic_ostream<Ch,Tr>& endl(basic_ostream<Ch,Tr>&) ; // put '\n' and flush
template <class Ch, class Tr>
 basic_ostream<Ch,Tr>& ends(basic_ostream<Ch,Tr>&) ; // put '\0' and flush
template <class Ch, class Tr>
 basic_ostream<Ch,Tr>& flush(basic_ostream<Ch,Tr>&) ; // flush stream
template <class Ch, class Tr>
 basic_istream<Ch,Tr>&ws(basic_istream<Ch,Tr>&) ; // eat whitespace
smanip resetiosflags(ios_base::fmtflags f) ; // clear flags
smanip setiosflags(ios_base::fmtflags f) ; // set flags
smanip setbase(int b) ; // output integers in base b
smanip setfill(int c) ; // make c the fill character
smanip setprecision(int n) ; // n digits after decimal point
smanip setw(int n) ; // next field is n char
```

Strumienie plikowe

- Oto pełen program do kopiowania jednego pliku do drugiego. Nazwy plików podaje się jako argumenty wywołania

```
#include <fstream>
#include <cstdlib>
void error(const char* p, const char* p2= "")
{
 cerr << p << ´ ´ << p2 << ´\n´;
 std::exit(1) ;
}
int main(int argc, char* argv[])
{
 if (argc != 3) error("wrong number of arguments") ;
 std::ifstream from(argv[1]) ; // open input file stream
 if (!from) error("cannot open input file",argv[1]) ;
 std::ofstream to(argv[2]) ; // open output file stream
 if (!to) error("cannot open output file",argv[2]) ;
 char ch;
 while (from.get(ch)) to.put(ch) ;
 if (!from.eof() || !to) error("something strange happened") ;
}
```

Strumienie plikowe

- Klasa *basic_ofstream* jest zadeklarowana w *<fstream>* następująco

```
template <class Ch, class Tr = char_traits<Ch> >
 class basic_ofstream : public basic_ostream<Ch,Tr> {
 public:
 basic_ofstream() ;
 explicit basic_ofstream(const char* p, openmode m = out) ;
 basic_filebuf<Ch,Tr>* rdbuf() const;
 bool is_open() const;
 void open(const char* p, openmode m = out) ;
 void close() ;
};
```

- Jak zwykle, dla najbardziej popularnych typów są dostępne definicje

```
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char> fstream;
typedef basic_ifstream<wchar_t>wifstream;
typedef basic_ofstream<wchar_t>wofstream;
typedef basic_fstream<wchar_t>wfstream;};
```

Strumienie plikowe

- Strumień *ifstream* jest podobny do *ofstream*, z tym, że jest wyprowadzony z *istream* i domyślnie otwarty do czytania. Ponadto biblioteka standardowa oferuje strumień *fstream*, który jest podobny do *ofstream*, ale jest wyprowadzony z *iostream* i domyślnie można z niego zarówno czytać, jak i do niego pisać.
- Konstruktory strumieni plikowych przyjmują drugi argument, specyfikujący różne tryby otwarcia

```
class ios_base {
 public:
 // ...
 typedef implementation_defined3 openmode;
 static openmode app, // append
   ate, // open and seek to end of file (pronounced ``at end'')
   binary, // I/O to be done in binary mode (rather than text mode)
   in, // open for reading
   out, // open for writing
   trunc; // truncate file to 0-length
 // ...
};
```

Strumienie napisowe

- Strumień można związać z napisem (*string*), tzn. można czytać z napisu i do niego pisać, używając udogodnień formatujących dostarczanych przez strumienie. Takie strumienie nazywają się *stringstream* i są zdefiniowane w *<sstream>*

```
template <class Ch, class Tr=char_traits<Ch> >
 class basic_stringstream : public basic_iostream<Ch,Tr> {
 public:
 explicit basic_stringstream(ios_base::openmode m = out|in) ;
 explicit basic_stringstream(const basic_string<Ch>& s, openmode m = out|in);
 basic_string<Ch> str() const; // get copy of string
 void str(const basic_string<Ch>& s) ; // set value to copy of s
 basic_stringbuf<Ch,Tr>* rdbuf() const;
};
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char> stringstream;
typedef basic_istringstream<wchar_t> wistringstream;
typedef basic_ostringstream<wchar_t> wostringstream;
typedef basic_stringstream<wchar_t> wstringstream;
```

Strumienie napisowe

- Można użyć *ostringstream* do formatowania komunikatów napisowych

```
extern const char* std_message[] ;
string compose(int n, const string& cs)
{
 ostringstream ost;
 ost<<"error("<< n << ") " <<std_message[n]<<" (user comment: "<<cs<< ´)´;
 return ost.str() ;
}
```

- Nie trzeba sprawdzać przepełnienia, ponieważ ost jest rozszerzane w miarę potrzeby.
- Można podać wartość początkową napisu w konstruktorze

```
string compose2(int n, const string& cs)
{
 ostringstream ost("error(",ios_base::ate) ;
 ost << n << ") " << std_message[n] << " (user comment: " << cs << ´)´;
 return ost.str() ;
}
```

Strumienie napisowe

- Klasa *istringstream* umożliwia czytanie strumienia wejściowego z napisu

```
#include <sstream>
void word_per_line(const string& s) // prints one word per line
{
 istringstream ist(s) ;
 string w;
 while (ist>>w) cout <<w << ´\n´;
}
int main()
{
 word_per_line("If you think C++ is difficult, try English") ;
}
```

- Napis będący inicjatorem jest kopiowany do *istringstream*. Koniec napisu kończy wejście.