



C++17 i STL

Lokalizacja



Lokalizacja



- ▶ Biblioteka standardowa C++ udostępnia środki umożliwiające tworzenie programów międzynarodowych – wpływają one głównie na wykorzystanie operacji wejścia-wyjścia oraz przetwarzania łańcuchów znakowych.
- ▶ W przypadku umiędzynarodawiania programów ważne są dwie związane z tym kwestie:
 - ▶ różne zestawy znaków mają różne właściwości;
 - ▶ użytkownik programu oczekuje zastosowania w nim narodowych lub kulturowych konwencji.
- ▶ Rozwiązaniem problemu umiędzynarodawiania programów jest zastosowanie obiektów ustawień lokalnych, reprezentujących rozszerzalny zbiór właściwości, które muszą zostać dostosowane do określonych konwencji lokalnych.



Kodowanie znaków

- ▶ W przypadku **reprezentacji wielobajtowej** (ang. multibyte representation) liczba bajtów wykorzystanych w celu określenia znaku jest zmienna.
- ▶ W przypadku **reprezentacji szerokiej** za pomocą zestawu znaków szerszego niż jeden bajt zakresu (ang. wide-character representation) liczba bajtów wykorzystana w celu określenia znaku jest zawsze jednakowa i nie zależy od reprezentowanego znaku.
- ▶ Reprezentacja wielobajtowa jest bardziej zwarta od reprezentacji szerokiej:
 - ▶ reprezentacja wielobajtowa jest zazwyczaj używana w celu przechowywania danych na zewnątrz programu;
 - ▶ znacznie prościej jest przetwarzać znaki stałego rozmiaru, dlatego też reprezentacja za pomocą zestawu znaków szerokiego zakresu jest zazwyczaj używana wewnątrz programów.



Zestawy znaków

- ▶ **US-ASCII** – 7-bitowy zestaw znaków ustandaryzowany już w 1963 roku dla dalekopisów i innych urządzeń.
- ▶ **ISO-Latin-1** lub **ISO-8859-1** oraz **ISO-Latin-2** lub **ISO-8859-2** – 8-bitowy zestaw znaków ustandaryzowany w roku 1987 na użytek języków z krajów Europy Zachodniej oraz Europy Środkowej.
- ▶ **UCS-2** – 16-bitowy zestaw znaków o stałym rozmiarze znaków, definiujący 65 536 najważniejszych znaków ze standardów Universal Character Set i Unicode.
- ▶ **UTF-8** – wielobajtowy zestaw znaków, z rozmiarem znaku od jednego do czterech 8-bitowych oktetów, odwzorowujący wszystkie znaki standardów Universal Character Set i Unicode (stosowany powszechnie w sieci WWW).
- ▶ **UTF-16** – wielobajtowy zestaw znaków o rozmiarze znaku od jednego do dwóch jednostek kodowych po 16 bitów każda, odwzorowujący wszystkie znaki standardów Universal Character Set i Unicode.
- ▶ **UCS-4** albo **UTF-32** – 32-bitowy zestaw znaków o stałym rozmiarze znaku, odwzorowujący wszystkie znaki standardów Universal Character Set i Unicode.

Zestawy znaków

	n	j	␣	␣	ä	+	€	␣	␣	1
ASCII (7-bitowy)	6E	6A	20	Brak	20	2B	20	Brak	20	31
ISO-8859-1 (8-bitowy)	6E	6A	20	E4	20	2B	20	Brak	20	31
ISO-8859-15 (8-bitowy)	6E	6A	20	E4	20	2B	20	A4	20	31
Windows-1252 (8-bitowy)	6E	6A	20	E4	20	2B	20	80	20	31
UTF-8	6E	6A	20	C3 A4	20	2B	20	E2 82 AC	20	31
UTF-16/UCS-2	006E	006A	0020	00E4	0020	002B	0020	20AC	0020	0031
UTF-32/UCS-4	0000006E	0000006A	00000020	000000E4	00000020	0000002B	00000020	000000AC	00000020	00000031
	n	j	␣	␣	ä	␣	␣			

Obsługa zestawów znaków

- ▶ **char** – nadaje się do obsługi wszystkich jednobajtowych (8-bitowych) zestawów znaków, jak US-ASCII, ISO-Latin-1, ISO-Latin-2, ISOLatin-9 i dodatkowo ze względu na specyfikę zestawu znaków typ `char` nadaje się też do reprezentowania oktetów zestawu znaków UTF-8.
- ▶ **char16_t** – (dostępny od C++11) nadaje się do reprezentowania znaków UCS-2 i do stosowania w roli jednostki kodowania UTF-16.
- ▶ **char32_t** – (dostępny od C++11) można używać do reprezentowania znaków i jednostek kodowania UCS-4/UTF-32.
- ▶ **wchar_t** – jest przeznaczony do reprezentowania wartości najszerszego zestawu znaków obsługiwanego w danej implementacji; zazwyczaj jest on równoważny typowi `char16_t` lub `char32_t`.


Literały napisowe

- ▶ Za pomocą przedrostka kodowania można określić kodowanie znaków odpowiednie dla literału napisowego – zdefiniowano następujące przedrostki kodowania:
 - ▶ **u8** – definiuje kodowanie UTF-8; literał napisowy UTF-8 jest inicjalizowany znakami zakodowanymi w łańcuchu znaków zgodnie z UTF-8; pojedyncze znaki literału są typu `char`.
 - ▶ **u** – określa literał napisowy w kodowaniu UTF-16, ze znakami typu `char16_t`.
 - ▶ **U** – określa literał napisowy w kodowaniu UTF-32, ze znakami typu `char32_t`.
 - ▶ **L** – określa literał napisowy w kodowaniu zależnym od implementacji, ze znakami typu `wchar_t`.
- ▶ Przedrostek kodowania może występować przed symbolem literału dosłownego **R**.
- ▶ Przykłady:
`L"ahoj"`
`uR"str(\ścieżka\plik.txt)str"`

Cechy znaków


- ▶ Klasy określające łańcuchy znakowe oraz strumienie przystosowane są do wykorzystywania standardowych typów wbudowanych, a zwłaszcza `char` oraz `wchar_t`, a od C++11 również `char16_t` oraz `char32_t`.
- ▶ Szczegóły dotyczące obsługi rzeczy zależnych od danej reprezentacji umieszczone są w dodatkowej oddzielnej klasie `char_traits<>` dla typów `char` i `wchar_t`, a od C++11 także dla typów `char16_t` i `char32_t`:

```
namespace std {  
    template<> struct char_traits<char>;  
    template<> struct char_traits<wchar_t>;  
    template<> struct char_traits<char16_t>;  
    template<> struct char_traits<char32_t>;  
}
```

Funkcje składowe klasy, określającej cechy znaków

Wyrażenie	Znaczenie
<code>char_type</code>	Typ znaku (argument szablonu dla klasy <code>char_traits</code>).
<code>int_type</code>	Typ na tyle duży, aby reprezentować dodatkową, nieużywaną w innym celu wartość określającą koniec pliku.
<code>pos_type</code>	Typ używany w celu reprezentacji pozycji w strumieniach.
<code>off_type</code>	Typ używany w celu reprezentacji przesunięcia pomiędzy pozycjami w strumieniach.
<code>state_type</code>	Typ używany w celu reprezentacji bieżącego położenia w strumieniach wielobajtowych.



Funkcje składowe klasy, określającej cechy znaków


Wyrażenie	Znaczenie
<code>assign(c1,c2)</code>	Przypisuje znak c2 do znaku c1.
<code>eq(c1,c2)</code>	Zwraca wartość logiczną, określającą, czy znaki c1 oraz c2 są jednakowe.
<code>lt(c1,c2)</code>	Zwraca wartość logiczną, określającą, czy znak c1 jest mniejszy od znaku c2.
<code>length(s)</code>	Zwraca długość łańcucha znakowego s.

Funkcje składowe klasy, określającej cechy znaków

Wyrażenie	Znaczenie
<code>compare(s1,s2,n)</code>	Porównuje do n znaków w łańcuchach s1 oraz s2.
<code>copy(s1,s2,n)</code>	Kopiuje n znaków z łańcucha s2 do łańcucha s1.
<code>move(s1,s2,n)</code>	Kopiuje n znaków z łańcucha s2 do łańcucha s1, przy czym łańcuchy s1 oraz s2 mogą na siebie zachodzić.
<code>assign(s,n,c)</code>	Przypisuje znak c do n znaków łańcucha s.
<code>find(s,n,c)</code>	Zwraca wskaźnik do pierwszego znaku w łańcuchu s, który jest równoważny znakowi c, lub zwraca wartość zero, jeśli znak ten nie zostanie odnaleziony wśród pierwszych n znaków.

Funkcje składowe klasy, określającej cechy znaków

Wyrażenie	Znaczenie
<code>eof()</code>	Zwraca wartość znaku, określającego znak końca pliku.
<code>to_int_type(c)</code>	Konwertuje znak <code>c</code> do odpowiadającej mu reprezentacji przy użyciu typu <code>int_type</code> .
<code>to_char_type(i)</code>	Konwertuje reprezentację znaku <code>i</code> określonego w postaci typu <code>int_type</code> do typu znakowego (wynik konwersji znaku EOF jest nieokreślony).
<code>not_eof(i)</code>	Zwraca wartość <code>i</code> , chyba że <code>i</code> jest wartością używaną do określenia znaku końca pliku (EOF). W takim przypadku zwracana jest wartość zależna od implementacji, różna od znaku EOF.
<code>eq_int_type(i1,i2)</code>	Sprawdza zgodność dwóch znaków <code>i1</code> oraz <code>i2</code> reprezentowanych w postaci wartości typu <code>int_type</code> (oznacza to, że znaki mogą być znakami końca pliku EOF).



Umiejdzynarodawianie znaków specjalnych

- ▶ Jednym z problemów związanych z kodowaniem znaków jest sposób umiejdzynarodawiania znaków nowego wiersza lub końca łańcucha znakowego.
- ▶ W klasie `basic_ios<>` zdefiniowane zostały funkcje składowe o nazwach `widen()` oraz `narrow()`, które mogą zostać wykorzystane w tym celu.
- ▶ Znak nowego wiersza lub znak końca łańcucha może zostać zapisany przy użyciu kodowania właściwego dla danego strumienia `strm`:

```
strm.widen('\n')  
strm.widen('\0')
```



Obiekty ustawień lokalnych


- ▶ Obiekty ustawień lokalnych (ang. locales), zawierają wszystkie konwencje narodowe oraz kulturowe.
- ▶ Format łańcucha znakowego definiującego obiekt ustawień lokalnych wygląda normalnie w następujący sposób:
język[_obszar[.kod]][@modyfikator]
- ▶ Przykłady:
pl_PL.ISO-8859-2
de_DE.utf8
en_GB

Wykorzystywanie ustawień lokalnych

- Zmiana ustawień lokalnych wpływa na wynik działania funkcji sortujących i manipulujących znakami, w rodzaju `isupper()` oraz `toupper()`, jak również funkcji wejścia-wyjścia, jak chociażby `printf()`.
- Ustalenie ustawień lokalnych realizuje funkcja `setlocale()`.
- Ustawienia lokalne zawarte są w obiekcie typu `locale`.

- Przykłady:


```
// klasyczne ustawienia języka C przy
// wczytywaniu danych ze standardowego wejścia
cin.imbue(locale::classic());
// ustawienia dla języka polskiego przy
// zapisie danych na standardowe wyjście
cout.imbue(locale("pl_PL"));
```



Wykorzystywanie ustawień lokalnych

- ▶ Jeśli program ma respektować lokalne konwencje, powinien w tym celu używać obiektów je określających.
- ▶ W celu zainstalowania globalnego obiektu określającego ustawienia lokalne może zostać wykorzystana funkcja składowa klasy `std::locale` o nazwie `global()`.
- ▶ Obiekt ten jest wtedy używany jako domyślna wartość funkcji przyjmujących w charakterze domyślnego argumentu obiekt klasy `locale`.
- ▶ Przykład:

```
std::locale::global(std::locale("pl_PL"));
```



Wykorzystywanie ustawień lokalnych

- ▶ Ustawienie globalnego obiektu ustawień lokalnych nie zastępuje ustawień regionalnych zachowanych w obiektach – modyfikuje ono jedynie obiekt `locale` skopiowany w momencie tworzenia go przy użyciu konstruktora domyślnego.
- ▶ Obiekty reprezentujące strumienie przechowują własne obiekty `locale`.
- ▶ Przykład:

```
std::cin.imbue(std::locale());  
std::cout.imbue(std::locale());  
std::cerr.imbue(std::locale());
```

Aspekty ustawień lokalnych

- ▶ Obiekt obsługujący określony aspekt ustawień lokalnych nazywany jest aspektem.
- ▶ Obiekt `locale` jest kontenerem różnych obiektów określających aspekty.
- ▶ W celu uzyskania dostępu do aspektu danych ustawień lokalnych w charakterze indeksu używany jest typ odpowiadającego mu obiektu `facet` – typ ten jest przekazywany jawnie jako argument konkretyzacji szablonu funkcji `use_facet()`, używającej określonego, żądanego aspektu.
- ▶ Przykład:

```
std::use_facet<std::num_punct<char>>(loc)
```

Aspekty ustawień lokalnych

Kategoria	Typ aspektu	Przeznaczenie
numeric	num_get<>()	Numeryczne dane wejściowe.
	num_put<>()	Numeryczne dane wyjściowe.
	numpunct<>()	Symbole używane w celu numerycznych operacji wejścia-wyjścia.
monetary	money_get<>()	Dane wejściowe określające jednostkę waluty.
	money_put<>()	Dane wyjściowe określające jednostkę waluty.
	moneypunct<>()	Symbole używane w celu operacji wejścia-wyjścia wykorzystujących jednostki waluty.

Aspekty ustawień lokalnych

Kategoria	Typ aspektu	Przeznaczenie
time	time_get<>()	Dane wejściowe określające czas i datę.
	time_put<>()	Dane wyjściowe określające czas i datę.
ctype	ctype<>()	Informacje o znakach (toupper(), isupper()).
	codecvt<>()	Konwersja pomiędzy różnymi standardami kodowania znaków.
collate	collate<>()	Sortowanie łańcuchów znakowych.
messages	messages<>()	Pobieranie łańcuchów znakowych określających komunikaty.



Klasa `locale`

- ▶ Klasa `locale` to niezmienny kontener przechowujący aspekty.
- ▶ Niezmiennność klasy `locale` polega na tym, że obiekty `facet` w nich przechowywane nie mogą być zmienione (z wyjątkiem operacji kopiowania obiektu `locale`).
- ▶ Odwołanie do obiektu reprezentującego aspekt, przechowywanego w obiekcie klasy `locale` następuje przy użyciu typu obiektu `facet` w charakterze indeksu.
- ▶ Każdy aspekt udostępnia inny interfejs i pasuje do innego zastosowania – w tym celu jest wykorzystywany typ przekazany w charakterze indeksu.

Kombinacje obiektów `locale`

- ▶ Klasa `locale` posiada zdefiniowany konstruktor kopiujący i przypisanie kopiujące.
- ▶ Konstruktor bezargumentowy tworzy kopie bieżącego globalnego obiektu `locale`.
- ▶ Można też utworzyć obiekt klasy `locale` podając konkretny symbol lokalizacji w konstruktorze (dla łańcucha pustego "" tworzony jest obiekt `locale` z ustawieniami macierzystymi dla środowiska wykonania).
- ▶ Specjalna wersja konstruktora `locale(loc1, loc2, cat)` tworzy kopię obiektu `loc1` ze wszystkimi aspektami z kategorii `cat` skopiowanymi z `loc2`.
- ▶ Konstruktor `locale(loc, fp)` tworzy kopię obiektu `loc` z dodanym aspektem wskazywanym przez `fp`.




Aspekty dla lokalizacji

- ▶ Funkcja o nazwie `use_facet()` zwraca referencję do obiektu `facet`. Typ tej referencji jest zgodny z typem przekazanym jawnie w charakterze argumentu szablonu. Jeśli obiekt `locale` przekazany jako argument nie posiada odpowiedniego aspektu, funkcja generuje wyjątek o nazwie `bad_cast`.
- ▶ Funkcja o nazwie `has_facet()` może być wykorzystana w celu sprawdzenia, czy określony aspekt jest obecny w przekazanym obiekcie `locale`.


Klasa `facet`

- ▶ Wszystkie obiekty `locale` muszą zawierać pewne standardowe obiekty `facet` reprezentujące aspekty.
- ▶ Użytkownik może również zainstalować swoje własne aspekty oraz zastąpić te już zdefiniowane.
- ▶ Klasa aspektu musi spełniać dwa wymagania:
 - ▶ dziedziczenie publiczne z klasy `std::locale::facet` – klasa bazowa definiuje głównie pewien mechanizm zliczania odwołań (referencji) używany wewnętrznie przez obiekty `locale`.
 - ▶ publiczna składowa statyczna o nazwie `id` typu `locale::id` – składowa ta jest wykorzystywana w celu wyszukania aspektu w obiekcie `locale` przy użyciu typu aspektu.
- ▶ Standardowe klasy aspektów spełniają dwa dodatkowe wymagania:
 - ▶ wszystkie funkcje składowe zadeklarowane są jako `const` – jest to przydatne, ponieważ funkcja `use_facet()` zwraca referencję do aspektu typu `const`;
 - ▶ wszystkie funkcje publiczne są niewirtualne i przekazują każde żądanie do chronionej funkcji wirtualnej.




Formatowanie wartości liczbowych

- ▶ Formatowanie wartości liczbowych przekształca wewnętrzną reprezentację liczb na odpowiadającą im reprezentację tekstową.
- ▶ Operatory strumieni przekazują rzeczywistą konwersję do aspektów kategorii `locale::numeric`. Kategoria ta jest utworzona przez trzy aspekty:
 - ▶ `num_punct`, obsługujący separatory dziesiętne używane podczas przetwarzania oraz formatowania wartości liczbowych;
 - ▶ `num_put`, obsługujący formatowanie wartości numerycznych;
 - ▶ `num_get`, obsługujący przetwarzanie wartości numerycznych – analiza łańcucha znakowego.



Formatowanie wartości liczbowych

- ▶ Funkcje składowe aspektu o nazwie `num_punct<>`:
 - ▶ `decimal_point()` – zwraca znak używany w charakterze separatora dziesiętnego;
 - ▶ `thousands_sep()` – zwraca znak używany w charakterze separatora rozdzielającego tysiące;
 - ▶ `grouping()` – zwraca łańcuch znakowy string opisujący pozycje separatorów rozdzielających setki tysięcy;
 - ▶ `truenamename()` – zwraca reprezentację tekstową wartości `true`;
 - ▶ `falsename()` – zwraca reprezentację tekstową wartości `false`.



Formatowanie wartości pieniężnych


- ▶ Kategoria `locale::monetary` składa się z trzech aspektów:
 - ▶ `moneypunct`, który definiuje format wartości jednostek walutowych;
 - ▶ `money_get` oraz `money_put`, które wykorzystują tę informację w celu formatowania lub analizy wartości walutowych.

Formatowanie wartości pieniężnych

- ▶ Funkcje składowe klasy aspektu `moneypunct<>`:
 - ▶ `decimal_point()` – zwraca znak używany w charakterze separatora dziesiętnego;
 - ▶ `thousands_sep()` – zwraca znak używany w charakterze separatora rozdzielającego tysiące;
 - ▶ `grouping()` – zwraca łańcuch znakowy string opisujący pozycje separatorów rozdzielających setki tysięcy;
 - ▶ `curr_symbol()` – zwraca łańcuch znakowy z symbolem waluty;
 - ▶ `positive_sign()` – zwraca łańcuch znakowy ze znakiem dodatnim;
 - ▶ `negative_sign()` – zwraca łańcuch znakowy ze znakiem ujemnym;
 - ▶ `frac_digits()` – zwraca liczbę cyfr dziesiętnych;
 - ▶ `pos_format()` – zwraca format używany z wartościami nieujemnymi;
 - ▶ `neg_format()` – zwraca format używany z wartościami ujemnymi.

Formatowanie czasu oraz daty

- ▶ Dwie klasy aspektów o nazwach `time_get` oraz `time_put` należące do kategorii `locale::time` udostępniają usługi analizy oraz formatowania czasu oraz daty.
- ▶ Służą do tego funkcje składowe operujące na obiektach typu `tm`. Typ ten jest zdefiniowany w pliku nagłówkowym o nazwie `<ctime>`. Obiekty nie są przekazywane bezpośrednio – w charakterze argumentu używany jest wskaźnik do nich.
- ▶ Łańcuch znakowy tworzony przez funkcję `strftime()` zależy od obiektu `locale`.



Klasyfikacja oraz konwersja znaków

- ▶ Biblioteka standardowa C++ definiuje dwa aspekty obsługujące znaki: `ctype` oraz `codecvt`. Oba należą do kategorii `locale::ctype`.
- ▶ Aspekt `ctype<>` jest używany głównie w celu klasyfikacji znaków, na przykład w przypadku sprawdzania, czy jest on literą. Udostępnia on również funkcje składowe służące do konwersji pomiędzy małymi a wielkimi literami, jak również pomiędzy typem `char` a typem znakowym, dla którego został utworzony egzemplarz aspektu.
- ▶ Aspekt `codecvt<>` jest wykorzystywany w celu konwersji pomiędzy różnymi standardami kodowania znaków i jest używany głównie przez obiekt `basic_filebuf` do konwersji pomiędzy reprezentacjami zewnętrznymi i wewnętrznymi.

Klasyfikacja znaków

- ▶ Usługi zdefiniowane w aspekcie `ctype<charT>`:
 - ▶ `is(m, c)` – sprawdza, czy znak `c` pasuje do maski `m`;
 - ▶ `is(beg, end, vec)` – dla każdego znaku z zakresu `[beg, end)` umieszcza w odpowiednim miejscu wektora `vec` maskę dopasowaną do znaku;
 - ▶ `scan_is(m, beg, end)` – zwraca wskaźnik do pierwszego znaku z zakresu `(beg, end)` dopasowanego do maski `m` lub w przypadku nie odnalezienia takiego znaku wartość `end`;
 - ▶ `scan_not(m, beg, end)` – zwraca wskaźnik do pierwszego znaku z zakresu `[beg, end)` który nie pasuje do maski `m`, lub w przypadku gdy wszystkie znaki pasują do maski – wartość `end`.

Klasyfikacja znaków

- ▶ Usługi zdefiniowane w aspekcie `ctype<charT>`:
 - ▶ `toupper(c)` – zwraca wielką literę odpowiadającą znakowi `c`, w przypadku gdy taka litera istnieje. W innym razie zwraca wartość `c`;
 - ▶ `toupper(begin, end)` – konwertuje każdą literę z zakresu pomiędzy `begin` a `end`, zamieniając ją wynikiem działania funkcji `toupper()`;
 - ▶ `tolower(c)` – zwraca małą literę odpowiadającą znakowi `c`, w przypadku gdy taka litera istnieje. W innym razie zwraca wartość `c`;
 - ▶ `tolower(begin, end)` – konwertuje każdą literę z zakresu pomiędzy `begin` a `end`, zamieniając ją wynikiem działania funkcji `tolower()`.

Klasyfikacja znaków


- ▶ Usługi zdefiniowane w aspekcie `ctype<charT>`:
 - ▶ `widen(c)` – zwraca znaku typu `char` przekonwertowany do typu `charT`;
 - ▶ `widen(begin, end, dest)` – dla każdego znaku z zakresu pomiędzy `begin` a `end` na odpowiednim miejscu argumentu `dest` umieszcza wynik działania funkcji `widen()`;
 - ▶ `narrow(c, default)` – zwraca znak typu `charT` przekonwertowany do typu `char` lub też domyślną wartość dla typu `char`, w przypadku gdy brak jest odpowiedniego znaku;
 - ▶ `narrow(begin, end, default, dest)` – dla każdego znaku z zakresu pomiędzy `begin` a `end` na odpowiednim miejscu argumentu `dest` umieszcza wynik działania funkcji `narrow()`.

Klasyfikacja znaków

- ▶ Funkcje globalne ułatwiające klasyfikację znaków:
 - ▶ `isalnum(c, loc)` – określa, czy znak `c` jest literą lub cyfrą (równoważna zapisowi postaci `isalpha() || isdigit()`);
 - ▶ `isalpha(c, loc)` – określa, czy znak `c` jest literą;
 - ▶ `isblank(c, loc)` – określa, czy znak `c` jest znakiem spacji lub tabulatora;
 - ▶ `isdigit(c, loc)` – określa, czy znak `c` jest cyfrą;
 - ▶ `isgraph(c, loc)` – określa, czy znak `c` jest znakiem możliwym do wyświetlenia innym niż spacja (równoważna zapisowi postaci `isalnum() || ispunct()`);
 - ▶ `ispunct(c, loc)` – określa, czy znak `c` jest znakiem przestankowym (to jest można go wyświetlić, lecz nie jest spacją, cyfrą ani literą);
 - ▶ `isspace(c, loc)` – określa, czy znak `c` jest znakiem spacji;
 - ▶ `isxdigit(c, loc)` – określa, czy znak `c` jest cyfrą w postaci szesnastkowej.

Klasyfikacja znaków

- ▶ Funkcje globalne ułatwiające klasyfikację znaków:
 - ▶ `isctrl(c, loc)` – określa, czy znak `c` jest znakiem kontrolnym;
 - ▶ `islower(c, loc)` – określa, czy znak `c` jest małą literą;
 - ▶ `isprint(c, loc)` – określa, czy znak `c` jest znakiem możliwym do wyświetlenia (włącznie ze spacjami);
 - ▶ `isupper(c, loc)` – określa, czy znak `c` jest wielką literą;
 - ▶ `tolower(c, loc)` – konwertuje znak `c` z małej litery na wielką;
 - ▶ `toupper(c, loc)` – konwertuje znak `c` z wielkiej litery na małą.



Konwersja standardów kodowania znaków

- ▶ W celu konwersji pomiędzy wewnętrznym a zewnętrznym standardem kodowania znaków używany jest aspekt o nazwie `codecv<>`. Może on zostać na przykład wykorzystany w celu konwersji pomiędzy standardem Unicode a EUC.
- ▶ Ta klasa aspektu wykorzystywana jest przez klasę `basic_filebuf` w celu dokonania konwersji pomiędzy reprezentacją wewnętrzną a reprezentacją znaków umieszczonych w pliku.



Sortowanie łańcuchów znakowych

- ▶ Aspekt o nazwie `collate<>` obsługuje różnice występujące pomiędzy konwencjami pod względem sortowania łańcuchów znakowych.
- ▶ Różne języki używają różnych zasad porównywania i sortowania dla określonych sekwencji znaków. Aspekt `collate` może zostać wykorzystany w celu udostępnienia sortowania łańcuchów w sposób oczekiwany przez użytkownika.

Sortowanie łańcuchów znakowych

- ▶ Funkcje składowe aspektu `collate<>`:
 - ▶ `compare (beg1, end1, beg2, end2)` – zwraca
 - ▶ 1 jeśli pierwszy łańcuch jest większy od drugiego,
 - ▶ 0 jeśli oba łańcuchy są jednakowe,
 - ▶ -1 jeśli pierwszy łańcuch jest mniejszy od drugiego;
 - ▶ `transform (beg, end)` – zwraca łańcuch znakowy porównywany z innymi przetransformowanymi łańcuchami;
 - ▶ `hash (beg, end)` – zwraca wartość skrótu (typu `long`) dla danego łańcucha znakowego.