Stromklassen als STL-Templates

Klassenhierachie

Die gebräuchlichen Stromklassen sind Templateinstanziierungen der Klassen `basic_istream`, `basic_ostream` bzw. `basic_iostream` bzgl. der Zeichentypen `char` ("narrow char") oder `wchar_t` ("wide char") und ihren Eigenschaften (traits) oder davon abgeleitet (Dateistromklassen, Stringstromklassen). Die üblichen kürzeren Namen werden per typedef eingeführt:

Klasse	*Aliasname*	*Zweck*
`basic_istream<char>`	`istream`	Standardeingabe
`basic_ostream<char>`	`ostream`	Standardausgabe
`basic_iostream<char>`	`iostream`	
`basic_ifstream<char>`	`ifstream`	Lesen von Dateien
`basic_ofstream<char>`	`ofstream`	Schreiben auf Dateien
`basic_fstream<char>`	`fstream`	Lesen- und Schreiben auf Dateien
`basic_istringstream<char>`	`istringstream`	Lesen von Strings
`basic_ostringstream<char>`	`ostringstream`	Schreiben auf Strings
`basic_stringstream<char>`	`stringstream`	Lesen- und Schreiben auf Strings
`basic_istream<wchar_t>`	`wistream`	Standardeingabe
`basic_ostream<wchar_t>`	`wostream`	Standardausgabe
`basic_iostream<wchar_t>`	`wiostream`	
`basic_ifstream<wchar_t>`	`wifstream`	Lesen von Dateien
`basic_ofstream<wchar_t>`	`wofstream`	Schreiben auf Dateien
`basic_fstream<wchar_t>`	`wfstream`	Lesen- und Schreiben auf Dateien
`basic_istringstream<wchar_t>`	`wistringstream`	Lesen von Strings
`basic_ostringstream<wchar_t>`	`wostringstream`	Schreiben auf Strings
`basic_stringstream<wchar_t>`	`wstringstream`	Lesen- und Schreiben auf Strings

Folgende Stromobjekte sind vordefiniert, zu Beginn des Programms geöffnet und mit Peripheriegeräten (Tastatur bzw. Bildschirm) verbunden.

Stromobjekt	*Stromklasse*	*Zweck*
`cin`	`istream`	Standardeingabe (gepuffert)
`cout`	`ostream`	Standardausgabe (gepuffert)
`cerr`	`ostream`	Standardfehlerausgabe (ungepuffert)
`clog`	`ostream`	Standardfehlerausgabe (gepuffert)
`wcin`	`istream`	Standardeingabe (gepuffert)
`wcout`	`ostream`	Standardausgabe (gepuffert)
`wcerr`	`ostream`	Standardfehlerausgabe (ungepuffert)
`wclog`	`ostream`	Standardfehlerausgabe (gepuffert)

Datei- und Stringstromobjekte werden durch Variablendefinition mit geeigneten Konstruktorargumenten angelegt. Nachträgliches Öffnen und Schließen von Dateien ist mittels Komponentenfunktionen (open, close) möglich.

Die Basisklassen sind wiederum abgeleitet vom Template `basic_ios`, das zeichenabhängigen Strom- und Formatierzustände enthält. Letzeres wiederum erbt von der Klasse `ios_base`, das die in der Regel zeichenunabhängigen Strom- und Formatierzustände beinhaltet. Auch hier gibt es wieder gängige Abkürzungen.

```
basic_ios<char>     ios
basic_ios<wchar_t>  wios
```

Aufgrund dieser Vererbungstruktur kann der Binärzugriff auf Dateien mit `ios::binary` (char) bzw. `wios::binary` (wchar_t) oder einheitlich mit `ios_base::binary` erfolgen. Dasselbe gilt auch für weitere Flags.

Unicode und UTF-8 unter Linux

Unicode ist ein international genormter 16-Bit-Zeichensatz, der mittlerweile zu einem 31-Bit-Zeichensatz (UCS – Universal Character Set) erweitert wurde. UCS soll die Schriftzeichen aller bekannten Sprachen enthalten.

Unter Linux kann für die interne Darstellung der erweiterte Zeichendatentyp `wchar_t` (wide char) verwendet werden. Zur Ein/Ausgabe, in Dateien und in Zeichen- und Zeichenkettenkonstanten ist unter Linux der Multibytezeichensatz UTF-8 gebräuchlich, der UCS-Zeichen auf Bytefolgen unterschiedlicher Länge abbildet. Bei Verwendung der korrekten Lokalisierung erfolgt die Umwandlung von der internen Darstellung in die externe und umgekehrt automatisch.

Literale für C-Zeichen und C-Zeichenkette basierend auf dem Datentyp `wchar_t` werden wie die entsprechenden Literale für char geschrieben, allerdings mit zusätzlich vorangegestelltem `L`.

Es ist auch möglich, in Zeichen- und Zeichenkettenkonstanten direkt die interne Kodierung zu verwenden: `\u`*xxxx* bzw. `\U`*xxxxxxxx*, worin *x* jeweils ein Hexadezimalziffer bezeichnet. Bestimmte Zeichen wie etwa ASCII-Zeichen und manche Kontrollzeichen sind davon allerdings ausgeschlossen.

Bsp.: Unicode-Zeichenkettenliterale (Ubuntu Linux 20.04, g++-9.4)

```
#include <iostream>
#include <locale>

using namespace std;

int main()
{
  locale::global(locale("de_DE.utf8"));  // Globale Voreinst. fuer Locale (und C-Locale)
  wcout.imbue(locale());                 // Locale fuer wcout aendern

  wcout << L"Umlaute:    \u00E4\u00F6\u00FC\u00C4\u00D6\u00DC" << endl;
  wcout << L"Kyrillisch: \u0414\u043E\u0431\u0440\u044B\u0439"
           L" \u0434\u0435\u043D\u044C" << endl;
  wcout << L"Chinesisch: \u4F60\u597D"  << endl;

  return 0;
}
```

Zeichenketten aus wchar_t

Aus Zeichen vom Datentyp `wchar_t` können C-Zeichenketten und C++-Strings gebildet werden.

C-Zeichenketten aus `wchar_t` enden mit `L'\0'` und werden mit den üblichen Ausnahmen in Ausdrücken in den Zeigerdatentyp `wchar_t *` umgesetzt.

Die C++-Stringklasse ist ein Template mit einem Zeichentypparameter, die beiden Templatespezialisierungen werden wie folgt abgekürzt:

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

Wegen der unterschiedlichen internen und externen Zeichendarstellung ist die Anzahl der Bytes und der Zeichen bei einer UTF-8-Textdatei normalerweise verschieden.

Bsp.: Zeilenweises Lesen und Ausgabe einer UTF-8 Textdatei (Ubuntu Linux 20.04, g++-9.4)

```
#include <iostream>
#include <fstream>
#include <locale>
#include <vector>
#include <string>

using namespace std;

int main()
{
  locale::global(locale("de_DE.utf8"));
  wcout.imbue(locale());

  wifstream ein("unicode.txt");
  ein.imbue(locale());
  wstring wstr;
  vector<wstring> zeilen;
  int nzeil=0, nwchar=0;

  while (getline(ein,wstr)) {
    zeilen.push_back(wstr);
    nzeil++;
    nwchar += wstr.size();
  }

  wcout << L"Anzahl der Zeilen:  " << nzeil  << endl
        << L"Anzahl der Zeichen: " << nwchar << endl; // ohne Zeilenenden

  for (int i=0; i<zeilen.size(); ++i)
    wcout << i+1 << L".Zeile: " << zeilen[i] << endl;

  return 0;
}
```

Sprachumgebungen (Locales)

In Sprachumgebungen (locales) sind regionale Unterschiede geregelt wie Sprache, Zeichensatz, Zeichenkettensortierung, Zahldarstellung, Datums- und Währungsformate. Sie sind in sogenannte „Facetten“ eingeteilt. Diese Facetten sind Klassen-Templates, über die mit bestimmten Funktionstemplates (`use_facet`) lesend zugegriffen werden kann. Für einige häufig gebrauchte Funktionstemplates gibt einfacher benutzbare Aufruffunktionen.

Testen und Umwandeln von Zeichen

Im folgenden ist c ein Zeichen (z.B. Typ `char` oder `wchar_t`) und `loc` ein mit dem Zeichentyp verträgliches Locale. Die Funktionen `is`*xxxxxx* liefern `bool` und `to`*xxxxxx* Zeichen.

Funktion	*Bedeutung*
`islower(c,loc)`	Kleinbuchstabe
`isupper(c,loc)`	Großbuchstabe
`isalpha(c,loc)`	Buchstabe
`isdigit(c,loc)`	Dezimalziffer
`isxdigit(c,loc)`	Hexadezimalziffer
`isalnum(c,loc)`	Buchstabe oder Ziffer
`isgraph(c,loc)`	druckbar ohne Leerzeichen
`isprint(c,loc)`	druckbar einschließlich Leerzeichen
`ispunct(c,loc)`	druckbar mit Ausnahme von Leerzeichen, Buchstaben und Ziffern
`isspace(c,loc)`	Zwischenraumzeichen
`iscntrl(c,loc)`	Steuerzeichen
`tolower(c,loc)`	Umwandlung in Kleinbuchstaben, liefert Zeichen
`toupper(c,loc`	Umwandlung in Großbuchstaben, liefert Zeichen

Ausgabe der druckbaren Unicode-Zeichen im Bereich $[M, N]$

```
#include <iostream>
#include <iomanip>
#include <locale>

using namespace std;

int main()
{
  locale::global(locale("de_DE.utf8"));
  locale loc;  // nach globaler Voreinstellung initialisiert
  wcin.imbue(loc); wcout.imbue(loc);

  int M,N; wcout << L"M N: "; wcin >> M >> N;

  wcout << L"   Dez   Hex     Zeichen" << endl << endl;
  for (int i=M; i<=N; ++i) {
    wcout << dec << setw(6) << i << hex << setw(6) << i << "     ";
    wchar_t c = i;
    if (isprint(c,loc)) wcout << c; else wcout << L"nicht druckbar";
    wcout << endl;
  }
  return 0;
}
```