Object-Oriented Programming VL

Laborprotokoll

Beispiel 4

Günther Neuwirth, Matr. Nr.0626638
e0626638@student.tuwien.ac.at
Manuel Mausz, Matr. Nr.0728348
manuel-tu@mausz.at

Wien, am 30. Mai 2009

Inhaltsverzeichnis

1 Aufgabenstellung - Beispiel 4

Beispielangaben zu
OOP (Objekt-Orientierte Programmierung)
SS 2009

Beispiel 4

Relevante Themen:

- Generizität
- Ableiten von generischer Klasse

Design und Implementierung:

Mit dieser Aufgabe soll speziell das Ableiten von generischen Klassen (Templates) geübt werden. Dazu sind die Klassen des CPU-Simulator myCPU aus Beispiel 3 als generische Klassen umzuschreiben, sodass anstelle von CDat eine alternative Klasse (mit beliebigen Rechenregeln und Zahlenformat) benutzt werden kann. Die Rechenregeln bestimmen die Berechnung neuer Werte und das Zahlenformat bestimmt die Parse-Funktion, mit welcher eine Zahl der Speicherbelegungsdatei zu interpretieren ist.

Neben der Klasse CDat sollen nun zwei weitere Klassen als Rechentypen implementiert werden:

- CDatSet:
  Diese Klasse benutzt als Rechenregeln analog wie CDat den C++ Datentyp int. Allerdings wird ein anderes Parserformat für die Speicherbelegungsdatei benutzt: eine Zahl n wird durch n-mal den Buchstaben 'o' (klein O) geschrieben. Damit können allerdings nur positive Zahlen dargestellt werden (negative Zahlen können erst durch anschließende Negation im Programm erreicht werden).
  Beispiel für eine Speicherbelegungsdatei mit den Zahlen 6 und 8:
  ooooo
  oooooooo
- CDatN
  Diese Klasse kann im Konstruktor mit einem Parameter int width initialisiert werden (der Default-Konstruktor ohne diesen Parameter ist zu deaktivieren, siehe Folien). Der Parameter width gibt an, mit wie vielen Bitstellen gerechnet werden soll, wobei die minimale gültige Stellenzahl 2 ist und die maximale Stellenzahl 32 ist. Wenn in der Speicherbelegungsdatei Zahlen stehen, die eine größere Stellenzahl als gegeben erfordern würde, so werden die höherwertigen Bits weggeschnitten. Die Rechenregeln sind auf die entsprechende Stellenzahl anzupassen.
  Beispiel: width=5 (5bits)
  Damit wird aus einer Zahl 150 (entspricht der Binärzahl 10010110) des Speicherbelegungsfile die Zahl 22 (entspricht der Binärzahl 10110). Die Beschränkung auf eine bestimmte Bitzahl kann in C++ ganz einfach mit den

Bitmanipulationsoperatoren gemacht werden. Beispielsweise kann die Zahl 150 folgendermaßen auf 5 Bits beschränkt werden: `var = ((1<<5)-1) & 150`.

Die Synopsis des Programms ist folgende:

`myCPU [-f⟨format⟩] -c ⟨programfile⟩ [-m ⟨memfile⟩]`

Das optionale Argument `–f⟨format⟩` wird benutzt, um das Rechenformat des CPU-Simulators umzustellen. Wird das Argument nicht angegeben, so soll die Rechenarithmetik von Beispiel 3 (Klasse CDat) bentutzt werden. Wird die Option angegeben, so hängt die zu wählende Rechenoperation vom Formatbezeichner. Ist der Formatbezeichner eine Zahl n von 2 bis 32, so soll die Klasse CDatN(n) benutzt werden. Ist der Formatbezeichner der Buchstabe ‚s', so soll die Klasse CDatSet als Rechenformat benutzt werden.

Die in der Spezifikation nicht genauer ausgeführten Teile können selbst sinnvoll ausgestaltet werden, wobei jedoch die folgenden Anforderungen einzuhalten sind.

Anforderungen:
Kommentieren Sie den Code, um das Verstehen des Codes zu vereinfachen. Zusätzlich soll zu Beginn jeder Klasse eine Klassenbeschreibung stehen. Am Beginn jeder Datei soll außerdem Name, MNr, KZ der Gruppenmitglieder sowie die Beispielnummer stehen. Ein nicht kommentierter Code wird negativ bewertet!

Ausnahmen (Exceptions) müssen immer abgefangen werden, sofern möglich. Überlegen Sie sich dabei aber auch, welche Ausnahmen zu einer Programmbeendigung führen müssen und in welchen Fällen es ausreicht, eine Warnung bzw. Fehlermeldung auszugeben. Bei jeder Ausnahmebehandlung ist auf jeden Fall eine Fehlermeldung auf die Fehlerausgabe (cerr) auszugeben. Als Orientierung seien einige mögliche Quellen für Ausnahmen aufgezählt: Speichermangel, inkorrekter Input, Schreiben auf Datei fehlgeschlagen, etc.

Schreiben Sie klare Zusicherungen an den passenden Stellen im Programm. Dort, wo man Zusicherungen kompakt als Code formulieren kann, wird empfohlen, die Zusicherungen zusätzlich als „assert()" hinzuschreiben, damit der Compiler Code generieren kann zur automatischen Prüfung der Zusicherungen zur Laufzeit. Mit der Compileroption „`-DNDEBUG`" kann man den fertig ausgetesteten Code generieren sodass „assert()" keinen Overhead mehr verursacht.

Benutzen Sie zur Programmentwicklung auch ein Makefile, welches zumindest die Targets „clean", „all" und „run" enthält. Mittels „make run" soll das Programm direkt aufgerufen werden können (mit sinnvollen Argumenten versehen).

Es sind die Programmierrichtlinien der LVA einzuhalten.

2 Beispiel 4

2.1 Design

Abbildung 1 zeigt das Klassendiagramm der Aufgabe.

Wie gefordert wurden die zusätzliche Datentypen CDatSet und CDatN implementiert und der Programmcode generisch gestaltet.

Dabei wurde CDatSet vom existierenden Typ CDat und boost::operators<CDatSet> mittels Mehrfachvererbung abgeleitet. So war es möglich die Implementierung, mittels Codewiederverwendung, auf das Überschreiben eines Kopierkonstruktors und des Operators operator»() zu beschränken.

Da der Typ CDatN Werte variabler Bitlänge repräsentiert, war es notwendig diesen vollständig neu zu implementieren. Es wurde von boost::operators<CDatN> abgeleitet und dabei in jeder Methode sicherstellt, dass der transportierte Wert immer auf die geforderte Bitanzahl beschränkt wird. Weiters wurde ein Kopierkonstruktor definiert, der einen Integerwert und einen vorzeichenlosen Integerwert als Argument erwartet. Letzterer ist mit dem Defaultwert 31 deklariert und repräsentiert die Bitanzahl, auf die Ersterer beschränkt wird. Damit ist es möglich alle vorhandenen Datentypen einheitlich zu verarbeiten. Der Defaultkonstruktor wurde wie gewünscht deaktiviert.

Um nun die Wahl des Datentyps zu ermöglichen, wurde die Synopsis des Programms mit “-f” erweitert. Der Parameter dieser Option muss “s” oder ein Wert zwischen “2” und “32” sein. Im Hauptprogramm wird anhand dieses Parameters der zuständige Datentyp instanziert und die Templatemethode cpu_run() aufgerufen, welche die Initialisierung und Abarbeitung des Programms anstößt.

Damit das Programm mit den verschiedenen Datentypen umgehen kann, wurden CCPU, CMem, CProgram, CInstuction und die abgeleiteten Instructions zu Templates umgewandelt, die als Template-Parameter den Typ des im Hauptprogramm instanzierten Datentyps, erwarten.

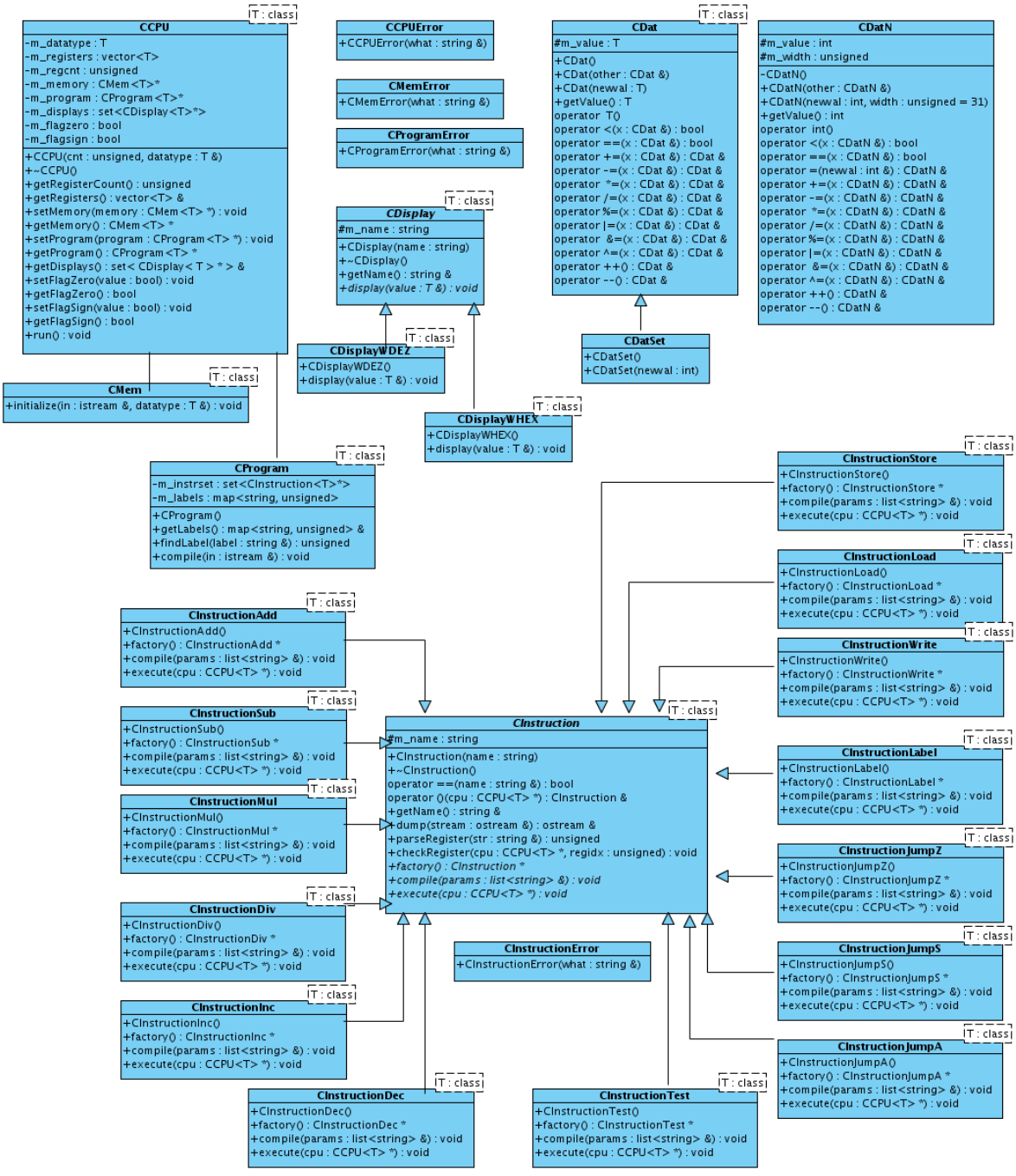

Abbildung 1: Klassendiagramm 1

2.2 Verwaltung der Ressourcen

Die Register von CCPU werden nun, anstatt in einem Array, in einem Vector verwaltet. Dieser wird mit der Anzahl der Register und dem, im Hauptprogramm, instanzierten Datentyp initialisiert. Dadurch ist das eigenhändige Löschen der Register im Destruktor nicht mehr notwendig, da dies der Destruktor des Vectors erledigt.

Siehe auch Punkt 3.1.

2.3 Fehlerbehandlung

Es wurden mehrere Exceptions von der Klasse std::invalid_argument abgeleitet. Diese sind CCPUError, CInstructionError, CMemError und CProgramError und werden von ihren zugehörigen Klassen entsprechend geworfen.

Wird zB.: eine Exception von Typ CInstructinError geworfen, so wird sie in CCPU zu CCPUError umgewandelt und an das Hauptprogramm weitergereicht, in dem wiederum eine Umwandlung in runtime_error() stattfindet. Das Fangen einer runtime_error() Exeption bewirkt eine fehlerbeschreibende Ausgabe auf std::cerr und den Programmabbruch mit Exitstatus 1.

2.4 Implementierung

Siehe Punkt 2.1 und Abbildung 1 sowie Punkt 4.

3 Projektverlauf

3.1 Probleme und Fallstricke

CDatN:

Mit der in der Angabe vorgeschlagenen Methode zur Begrenzung der Bitanzahl, ist es nur möglich positive Zahlen zu verarbeiten. Somit entspricht -1 dem Maximum der, mit der Bitanzahl, darstellbaren positiven Zahl usw..

Ein weiteres Problem besteht darin, wenn Register nicht mehr gelesen werden können, weil der Index des Registers ausserhalb des Bereichs des Datentyps liegt. In diesen Fall ist das Programmverhalten undefiniert.

Auf Grund der Forderung, den Defaultkonstruktor von CDatN zu deaktivieren, traten einige Probleme auf. So war es beispielsweise nicht weiter möglich, die Register von CCPU in einem Array zu organisieren, da zur Deklaration ein Defaultkonstruktor notwendig ist. Weiters musste eine Alternative zum nicht mehr anwendbaren boost::lexical_cast in CMem implementiert werden.

3.2 Arbeitsaufwand

Entwicklungsschritt / Meilenstein	Arbeitsaufwand
Erstes Design	1 Tage
Implementierung	2 Tage
Dokumentation (Doxygen) und Überprüfung aller Anforderungen gemäß der Programmierrichtlinien	4 Stunden
Erstellung des Protokolls	2 Stunden

4 Listings

4.1 mycpu.cpp

```
/**
 * @module mycpu
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  mycpu executes a programfile (in simple assembler) by parsing the
 *         programfile first. This creates a vector of instructions, which will
 *         be executed in linear order (except jumps) afterwards. In order to
 *         initialize the memory of the cpu before execution an optional
 *         memoryfile can be passed as commandline option.
 * @date   26.05.2009
 * @par Exercise
 *      4
 */

#include <boost/program_options.hpp>
#include <boost/lexical_cast.hpp>
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <stdlib.h>
#include "cdat.h"
#include "cdatset.h"
#include "cdatn.h"
#include "ccpu.h"
#include "cmem.h"
#include "cprogram.h"

#define REGISTERS 256

using namespace std;
namespace po = boost::program_options;

/* TODO */
template<class T>
void cpu_run(string& me, po::variables_map& vm, unsigned registers, T& datatype)
{
  CMem<T> memory;
  /* optionally initialize memory from file */
  if (vm.count("memory"))
  {
    string memoryfile(vm["memory"].as<string>());
    ifstream file(memoryfile.c_str(), ios::in);
    if (!file.is_open())
      throw runtime_error("Unable to open memoryfile '" + memoryfile + "' for
          reading.");

    try
    {
      memory.initialize(file, datatype);
      file.close();
    }
    catch(CMemError& ex)
    {
      file.close();
      std::stringstream sstr;
      sstr << "Error while reading from memoryfile:" << endl << "  " << ex.what();
      throw runtime_error(sstr.str());
    }
```

```
#if DEBUG
    memory.dump(cerr);
#endif
  }

  /* create program instance */
  CProgram<T> program;
  string programfile(vm["compile"].as<string>());
  ifstream file(programfile.c_str(), ios::in);
  if (!file.is_open())
    throw runtime_error("Unable␣to␣open␣programfile␣'" + programfile + "'␣for␣
        reading.");

  try
  {
    program.compile(file);
    file.close();
  }
  catch(CProgramError& ex)
  {
    file.close();
    std::stringstream sstr;
    sstr << "Error␣while␣compiling␣programfile:" << endl << "␣␣" << ex.what();
    throw runtime_error(sstr.str());
  }

#if DEBUG
  program.dump(cerr);
#endif

  /* execute the program */
  CCPU<T> cpu(registers, datatype);
  try
  {
    cpu.setMemory(&memory);
    cpu.setProgram(&program);
    cpu.run();
#if DEBUG
    //cpu.dumpRegisters(cerr);
#endif
  }
  catch(CCPUError& ex)
  {
    std::stringstream sstr;
    sstr << "Error␣while␣executing␣program:" << endl << "␣␣" << ex.what();
#if DEBUG
    memory.dump(cerr);
#endif
    throw runtime_error(sstr.str());
  }
}

/**
 * @func    main
 * @brief   program entry point
 * @param   argc   standard parameter of main
 * @param   argv   standard parameter of main
 * @return 0 on success, not 0 otherwise
 * @globalvars none
 * @exception   none
 * @pre   none
 * @post terminate with 0 if success else 1.
 *
 * parse commandline options, create and initialize memory,
```

```
 * create cprogram instance, which parses the programfile and
 * execute CCPU::run()
 * On error print error message to stderr.
 * Unknown commandline options will print a usage message.
 */
int main(int argc, char* argv[])
{
  string me(argv[0]);

  /* define commandline options */
  po::options_description desc("Allowed␣options");
  desc.add_options()
    ("help,h",     "this␣help␣message")
    ("format,f",   po::value<string>(), "input␣format")
    ("compile,c",  po::value<string>(), "input␣programfile")
    ("memory,m",   po::value<string>(), "input␣memoryfile");

  /* parse commandline options */
  po::variables_map vm;
  try
  {
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
  }
  catch(po::error& ex)
  {
    cerr << me << ":␣Error:␣" << ex.what() << endl;
    return 1;
  }

  /* print usage upon request or missing params */
  if (vm.count("help") || !vm.count("compile"))
  {
    cout << "Usage:␣" << me << "␣[-f␣<format>]␣-c␣<programfile>␣[-m␣<memoryfile>]"
        << endl;
    cout << desc << endl;
    return 0;
  }

  /* create memory, program and cpu from templates */
  try
  {
    if (vm.count("format"))
    {
      string format(vm["format"].as<string>());
      if (format == "s")
      {
        CDatSet datatype(0);
        cpu_run<CDatSet>(me, vm, REGISTERS, datatype);
      }
      else
      {
        unsigned bc;
        try
        {
          bc = boost::lexical_cast<unsigned>(format);
        }
        catch(boost::bad_lexical_cast& ex)
        {
          cerr << me << ":␣Paramater␣'format'␣has␣invalid␣or␣unknown␣format." <<
              endl;
          return 1;
        }
```

```
        if (bc < 2 || bc > 32)
        {
          cerr << me << ": Paramater 'format' must be inbetween 2 and 32." << endl;
          return 1;
        }

        CDatN datatype(0, bc);
        cpu_run<CDatN>(me, vm, REGISTERS, datatype);
      }
    }
    else
    {
      CDat<int> datatype(0);
      cpu_run<CDat<int> >(me, vm, REGISTERS, datatype);
    }
  }
  catch(runtime_error& ex)
  {
    cerr << me << ": " << ex.what() << endl;
    return 1;
  }

  return 0;
}

/* vim: set et sw=2 ts=2: */
```

4.2 cdat.h

```
/**
 * @module cdat
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Datatype template and datatype definition for CCPU and CMem
 * @date   26.05.2009
 */

#ifndef CDAT_H
#define CDAT_H 1

#include <boost/operators.hpp>
#include <iostream>

/**
 * @class CDat
 *
 * Datatype template for CCPU and CMem.
 */
template <class T>
class CDat
  : public boost::operators<CDat<T> >
{
  public:
    /**
     * @method CDat
     * @brief  Default ctor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  bad_alloc
     * @pre  none
     * @post none
     */
    CDat()
    {}

    /**
     * @method ~CDat
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    virtual ~CDat()
    {}

    /**
     * @method CDat
     * @brief  Copy constructor for CDat
     * @param  other  reference to CDat which will be copied
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CDat(const CDat& other)
      : m_value(other.m_value)
```

```
{}

/**
 * @method CDat
 * @brief  Copy constructor for T
 * @param  newval  new value for CDat
 * @return -
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat(const T newval)
  : m_value(newval)
{}

/**
 * @method getValue
 * @brief  returns value of CDat
 * @param  -
 * @return value of CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
T getValue() const
{
  return m_value;
}

/**
 * @method operator T
 * @brief  convert to T
 * @param  -
 * @return T
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
operator T()
{
  return m_value;
}

/**
 * @method operator<
 * @brief  implementation of operator <
 * @param  x  reference to CDat
 * @return true if cdat is less than object x
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
bool operator<(const CDat& x) const
{
  return m_value < x.m_value;
}

/**
 * @method operator==
 * @brief  implementation of operator ==
```

```
 * @param  x  reference to CDat
 * @return true if cdat equals object x
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
bool operator==(const CDat& x) const
{
  return m_value == x.m_value;
}

/**
 * @method operator+=
 * @brief  implementation of operator +=
 * @param  x  reference to CDat
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator+=(const CDat& x)
{
  m_value += x.m_value;
  return *this;
}

/**
 * @method operator-=
 * @brief  implementation of operator -=
 * @param  x  reference to CDat
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator-=(const CDat& x)
{
  m_value -= x.m_value;
  return *this;
}

/**
 * @method operator*=
 * @brief  implementation of operator *=
 * @param  x  reference to CDat
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator*=(const CDat& x)
{
  m_value *= x.m_value;
  return *this;
}

/**
 * @method operator/=
 * @brief  implementation of operator /=
 * @param  x  reference to CDat
```

```
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator/=(const CDat& x)
{
  m_value /= x.m_value;
  return *this;
}

/**
 * @method operator%=
 * @brief  implementation of operator %=
 * @param  x  reference to CDat
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator%=(const CDat& x)
{
  m_value %= x.m_value;
  return *this;
}

/**
 * @method operator|=
 * @brief  implementation of operator |=
 * @param  x  reference to CDat
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator|=(const CDat& x)
{
  m_value |= x.m_value;
  return *this;
}

/**
 * @method operator&=
 * @brief  implementation of operator &=
 * @param  x  reference to CDat
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator&=(const CDat& x)
{
  m_value &= x.m_value;
  return *this;
}

/**
 * @method operator^=
 * @brief  implementation of operator ^=
 * @param  x  reference to CDat
```

```
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator^=(const CDat& x)
{
  m_value ^= x.m_value;
  return *this;
}

/**
 * @method operator++
 * @brief  implementation of operator ++
 * @param  -
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator++()
{
  m_value++;
  return *this;
}

/**
 * @method operator--
 * @brief  implementation of operator --
 * @param  -
 * @return refecence to CDat
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDat& operator--()
{
  m_value--;
  return *this;
}

/**
 * @method operator<<
 * @brief  Shift/output operator for outputstream
 * @param  stream  reference to outputstream
 * @param  cdat    object which will be printed to stream
 * @return reference to outputstream
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
friend std::ostream& operator<<(std::ostream& stream, CDat cdat)
{
  stream << cdat.m_value;
  return stream;
}

/**
 * @method operator>>
 * @brief  Shift/read operator for inputstream
```

```
     * @param  stream  reference to inputstream
     * @param  cdat    reference to object which will be read from stream
     * @return reference to inputstream
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    friend std::istream& operator>>(std::istream & stream, CDat& cdat)
    {
      stream >> cdat.m_value;
      return stream;
    }

  protected:
    /* members */
    /** internal value of datatype */
    T m_value;
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.3 cdatset.h

```
/**
 * @module cdatset
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Datatype template and datatype definition for CCPU and CMem
 * @date   26.05.2009
 */

#ifndef CDATSET_H
#define CDATSET_H 1

#include <iostream>
#include "cdat.h"

/**
 * @class CDatSet
 *
 * Datatype template for CCPU and CMem.
 */
class CDatSet
  : public CDat<int>, public boost::operators<CDatSet>
{
  public:
    /**
     * @method CDatSet
     * @brief  Default ctor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  bad_alloc
     * @pre  none
     * @post none
     */
    CDatSet()
    {}

    /**
     * @method CDatSet
     * @brief  Copy constructor for int
     * @param  newval  new value for CDatSet
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CDatSet(const int newval)
      : CDat<int>(newval)
    {}

    /**
     * @method operator>>
     * @brief  Shift/read operator for inputstream
     * @param  stream  reference to inputstream
     * @param  cdat    reference to object which will be read from stream
     * @return reference to inputstream
     * @globalvars none
     * @exception  none
     * @pre  stream != null
     * @post cdat.m_value = count of char 'o'
     */
    friend std::istream& operator>>(std::istream & stream, CDatSet& cdat)
```

```
    {
      unsigned count = 0;
      while(stream.good() && !stream.eof())
      {
        int val = stream.get();
        if (val != 'o')
          break;
        ++count;
      }
      stream.clear();
      cdat.m_value = count;
      return stream;
    }
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.4 cdatn.h

```
/**
 * @module cdatn
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Datatype template and datatype definition for CCPU and CMem
 * @date   26.05.2009
 */

#ifndef CDATN_H
#define CDATN_H 1

#include <boost/operators.hpp>
#include <iostream>

/**
 * @class CDatN
 *
 * Datatype template for CCPU and CMem.
 */
class CDatN
  : public boost::operators<CDatN>
{
  private:
    /**
     * @method CDatN
     * @brief  Default ctor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  bad_alloc
     * @pre  none
     * @post none
     */
    CDatN()
    {}

  public:
    /**
     * @method ~CDatN
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    virtual ~CDatN()
    {}

    /**
     * @method CDatN
     * @brief  Copy constructor for CDatN
     * @param  other  reference to CDatN which will be copied
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CDatN(const CDatN& other)
      : m_value(other.m_value), m_width(other.m_width)
```

```
{}

/**
 * @method CDatN
 * @brief  Copy constructor for int
 * @param  newval  new value for CDatN
 * @param  width   maximum width
 * @return -
 * @globalvars none
 * @exception  std::runtime_error
 * @pre  none
 * @post none
 */
CDatN(const int newval, unsigned width = 31)
  : m_value(((1 << width) - 1) & newval), m_width(width)
{
  if (width < 2 || width > 32)
    throw std::runtime_error("width must be between 2 and 32");
}

/**
 * @method getValue
 * @brief  returns value of CDatN
 * @param  -
 * @return value of CDatN
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
int getValue() const
{
  return m_value;
}

/**
 * @method operator int
 * @brief  convert to int
 * @param  -
 * @return int
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
operator int()
{
  return m_value;
}

/**
 * @method operator<
 * @brief  implementation of operator <
 * @param  x  reference to CDatN
 * @return true if cdat is less than object x
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
bool operator<(const CDatN& x) const
{
  return m_value < x.m_value;
}
```

```
/**
 * @method operator==
 * @brief  implementation of operator ==
 * @param  x  reference to CDatN
 * @return true if cdat equals object x
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
bool operator==(const CDatN& x) const
{
  return m_value == x.m_value;
}

/**
 * @method operator=
 * @brief  implementation of operator =
 * @param  newval  reference to int
 * @return refecence to int
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDatN &operator=(const int& newval)
{
  m_value = ((1 << m_width) - 1) & newval;
  return *this;
}

/**
 * @method operator+=
 * @brief  implementation of operator +=
 * @param  x  reference to CDatN
 * @return refecence to CDatN
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDatN& operator+=(const CDatN& x)
{
  m_value = ((1 << m_width) - 1) & (m_value + x.m_value);
  return *this;
}

/**
 * @method operator-=
 * @brief  implementation of operator -=
 * @param  x  reference to CDatN
 * @return refecence to CDatN
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDatN& operator-=(const CDatN& x)
{
  m_value = ((1 << m_width) - 1) & (m_value - x.m_value);
  return *this;
}
```

```
/**
 * @method operator*=
 * @brief  implementation of operator *=
 * @param  x  reference to CDatN
 * @return refecence to CDatN
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDatN& operator*=(const CDatN& x)
{
  m_value = ((1 << m_width) - 1) & (m_value * x.m_value);
  return *this;
}

/**
 * @method operator/=
 * @brief  implementation of operator /=
 * @param  x  reference to CDatN
 * @return refecence to CDatN
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDatN& operator/=(const CDatN& x)
{
  m_value = ((1 << m_width) - 1) & (m_value / x.m_value);
  return *this;
}

/**
 * @method operator%=
 * @brief  implementation of operator %=
 * @param  x  reference to CDatN
 * @return refecence to CDatN
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDatN& operator%=(const CDatN& x)
{
  m_value = ((1 << m_width) - 1) & (m_value % x.m_value);
  return *this;
}

/**
 * @method operator|=
 * @brief  implementation of operator |=
 * @param  x  reference to CDatN
 * @return refecence to CDatN
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CDatN& operator|=(const CDatN& x)
{
  m_value = ((1 << m_width) - 1) & (m_value | x.m_value);
  return *this;
}
```

```
/**
 * @method operator&=
 * @brief implementation of operator &=
 * @param x reference to CDatN
 * @return refecence to CDatN
 * @globalvars none
 * @exception none
 * @pre none
 * @post none
 */
CDatN& operator&=(const CDatN& x)
{
  m_value = ((1 << m_width) - 1) & (m_value & x.m_value);
  return *this;
}

/**
 * @method operator^=
 * @brief implementation of operator ^=
 * @param x reference to CDatN
 * @return refecence to CDatN
 * @globalvars none
 * @exception none
 * @pre none
 * @post none
 */
CDatN& operator^=(const CDatN& x)
{
  m_value = ((1 << m_width) - 1) & (m_value ^ x.m_value);
  return *this;
}

/**
 * @method operator++
 * @brief implementation of operator ++
 * @param -
 * @return refecence to CDatN
 * @globalvars none
 * @exception none
 * @pre none
 * @post none
 */
CDatN& operator++()
{
  m_value = ((1 << m_width) - 1) & (m_value + 1);
  return *this;
}

/**
 * @method operator--
 * @brief implementation of operator --
 * @param -
 * @return refecence to CDatN
 * @globalvars none
 * @exception none
 * @pre none
 * @post none
 */
CDatN& operator--()
{
  m_value--;
  return *this;
}
```

```
    /**
     * @method operator<<
     * @brief  Shift/output operator for outputstream
     * @param  stream  reference to outputstream
     * @param  cdat    object which will be printed to stream
     * @return reference to outputstream
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    friend std::ostream& operator<<(std::ostream& stream, CDatN cdat)
    {
      stream << cdat.m_value;
      return stream;
    }

    /**
     * @method operator>>
     * @brief  Shift/read operator for inputstream
     * @param  stream  reference to inputstream
     * @param  cdat    reference to object which will be read from stream
     * @return reference to inputstream
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    friend std::istream& operator>>(std::istream & stream, CDatN& cdat)
    {
      stream >> cdat.m_value;
      cdat.m_value = ((1 << cdat.m_width) - 1) & cdat.m_value;
      return stream;
    }

  protected:
    /* members */
    /** internal value of datatype */
    int m_value;
    /** width of datatype */
    unsigned m_width;
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.5 cmem.h

```
/**
 * @module cmem
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Memory template and memory definition for CCPU
 * @date   26.05.2009
 */

#ifndef CMEM_H
#define CMEM_H 1

#include <vector>
#include <istream>
#include <sstream>
#include <stdexcept>
#ifdef DEBUG
# include <iostream>
# include <iomanip>
#endif

/**
 * @class CMemError
 *
 * Exception thrown by implemententations of CMem
 */
class CMemError
 : public std::invalid_argument
{
  public:
    /**
     * @method CMemError
     * @brief  Default exception ctor
     * @param  what  message to pass along
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CMemError(const std::string& what)
      : std::invalid_argument(what)
    {}
};

/**
 * @class CMem
 *
 * Extends std::vector template for use as memory for CCPU.
 */
template <class T>
class CMem
  : public std::vector<T>
{
  typedef std::vector<T> super;
  typedef typename super::iterator iterator;
  using super::size;
  using super::begin;
  using super::end;

  public:
    /**
     * @method initialize
```

```
     * @brief  initialize the vector with the content of istream. istream is
     *         read per line. empty lines will add unitialized elements.
     * @param  in         inputstream to read from
     * @param  datatype   reference instance of datatype to copy from
     * @return void
     * @globalvars none
     * @exception  CMemError
     * @pre  none
     * @post none
     */
    void initialize(std::istream& in, T& datatype)
    {
      if (!in.good())
        return;

      std::string line;
      unsigned i = 0;
      while (!in.eof() && in.good())
      {
        ++i;
        std::getline(in, line);

        /* skip last line if it's empty */
        if (line.empty() && in.eof())
          break;

        T value(datatype);
        if (!line.empty())
        {
          /* simple boost::lexical_cast replacement */
          std::stringstream interpreter;
          if(!(interpreter << line && interpreter >> value && interpreter.get() ==
               std::char_traits<char>::eof()))
          {
            std::stringstream sstr;
            sstr << "Unable to convert input (line " << i << ") to datatype";
            throw CMemError(sstr.str());
          }
        }

        push_back(value);
      }
    }

#if DEBUG
    /**
     * @method dump
     * @brief  dumps contents of vector to outputstream
     * @param  out  outputstream to write to
     * @return void
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    void dump(std::ostream& out)
    {
      out << "[MEMORY DUMP]" << std::endl;
      unsigned i = 0;
      for(iterator it = begin(); it != end(); ++it)
      {
        out << "[" << std::setw(4) << std::setfill('0') << i << "]  "
            << *it << std::endl;
        ++i;
```

```
        }
    }
#endif
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.6 cinstruction.h

```
/**
 * @module cinstruction
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Abstract class for instructions
 * @date   26.05.2009
 */

#ifndef CINSTRUCTION_H
#define CINSTRUCTION_H 1

#include <iostream>
#include <list>
#include <sstream>
#include <boost/lexical_cast.hpp>
#include <assert.h>
#include <stdexcept>

/**
 * @class CInstructionError
 *
 * Exception thrown by implemententations of CInstruction
 */
class CInstructionError
 : public std::invalid_argument
{
  public:
    /**
     * @method CInstructionError
     * @brief  Default exception ctor
     * @param  what  message to pass along
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CInstructionError(const std::string& what)
      : std::invalid_argument(what)
    {}
};

#include "ccpu.h"

/* forward declare CCPU */
template <class T>
class CCPU;

/**
 * @class CInstruction
 *
 * Abstract class for instructions
 */
template <class T>
class CInstruction
{
  public:
    /**
     * @method CInstruction
     * @brief  Default ctor
     * @param  name  name of instruction
     * @return -
```

```
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CInstruction(std::string name)
  : m_name(name)
{}

/**
 * @method ~CInstruction
 * @brief  Default dtor
 * @param  -
 * @return -
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
virtual ~CInstruction()
{}

/**
 * @method operator==
 * @brief  implementation of operator ==
 * @param  name reference to std::string
 * @return true if instructionname is name
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
virtual bool operator==(std::string& name)
{
  return name == m_name;
}

/**
 * @method operator()
 * @brief  implementation of operator (CCPU)
 * @param  cpu  pointer to cpu
 * @return -
 * @globalvars none
 * @exception  CInstructionError
 * @pre  none
 * @post none
 */
virtual CInstruction& operator()(CCPU<T> *cpu)
{
  execute(cpu);
  return *this;
}

/**
 * @method getName
 * @brief  returns instruction name
 * @param  -
 * @return name of instruction
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
virtual const std::string& getName()
```

```
{
  return m_name;
}

/**
 * @method dump
 * @brief  dumps information about instruction to outputstream
 * @param  stream  outputstream
 * @return reference to outputstream
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
virtual std::ostream& dump(std::ostream& stream)
{
  stream << m_name;
  return stream;
}

/**
 * @method operator<<
 * @brief  Shift/output operator for outputstream
 * @param  stream  reference to outputstream
 * @param  instr   object which will be printed to stream
 * @return reference to outputstream
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
friend std::ostream& operator<<(std::ostream& stream, CInstruction& instr)
{
  return instr.dump(stream);
}

/**
 * @method parseRegister
 * @brief  parses register syntax Rx (e.g. "R1")
 * @param  str  register in assembler syntax
 * @return registernumber
 * @globalvars none
 * @exception  CInstructionError
 * @pre  none
 * @post none
 */
virtual const unsigned parseRegister(const std::string& str);

/**
 * @method checkRegister
 * @brief  performs a register boundary check
 *         does the register exist in cpu?
 * @param  cpu     pointer to cpu
 * @param  regidx  registernumber
 * @return -
 * @globalvars none
 * @exception  CInstructionError
 * @pre  none
 * @post none
 */
virtual void checkRegister(CCPU<T> *cpu, const unsigned regidx);

/**
 * @method factory
```

```
     * @brief  creates a new instance of this instruction
     * @param  -
     * @return new instruction instance
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    virtual CInstruction *factory() = 0;

    /**
     * @method compile
     * @brief  parses instruction parameters and prepares the
     *         instruction for executing
     * @param  params  list of parameters of this instruction
     * @return -
     * @globalvars none
     * @exception  CInstructionError
     * @pre  none
     * @post none
     */
    virtual void compile(std::list<std::string>& params) = 0;

    /**
     * @method execute
     * @brief  executes the instruction
     * @param  cpu  pointer to cpu
     * @return -
     * @globalvars none
     * @exception  CInstructionError
     * @pre  none
     * @post none
     */
    virtual void execute(CCPU<T> *cpu) = 0;

  protected:
    /* members */
    /** name of instruction */
    std::string m_name;
};

/*----------------------------------------------------------------------------*/

template<class T>
const unsigned CInstruction<T>::parseRegister(const std::string& str)
{
  unsigned reg;
  if (str.length() < 2 || str[0] != 'r')
    throw CInstructionError("Invalid syntax of register");

  try
  {
    reg = boost::lexical_cast<unsigned>(str.substr(1));
  }
  catch(boost::bad_lexical_cast& ex)
  {
    throw CInstructionError("Invalid syntax of register");
  }

  return reg;
}

/*----------------------------------------------------------------------------*/
```

```
template<class T>
inline void CInstruction<T>::checkRegister(CCPU<T> *cpu, const unsigned regidx)
{
  assert(cpu != NULL);
  if (regidx >= cpu->getRegisterCount())
  {
    std::stringstream sstr;
    sstr << "Register R" << regidx << " doesn't exist (out of bound)";
    throw CInstructionError(sstr.str());
  }
}

#endif

/* vim: set et sw=2 ts=2: */
```

4.7 instructions.h

```
/**
 * @module instructions
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Implementations of CInstruction
 * @date   26.05.2009
 */

#ifndef INSTRUCTIONS_H
#define INSTRUCTIONS_H 1

#include "cinstruction.h"
#include "ccpu.h"
#include "cprogram.h"

/**
 * @class CInstructionInc
 *
 * Implementation of assembler command "inc"
 * Syntax: inc R1
 *   (R1++)
 */
template <class T>
class CInstructionInc
  : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionInc()
      : CInstruction<T>("inc")
    {}

    CInstructionInc *factory()
    {
      return new CInstructionInc;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
};

/*-------------------------------------------------------------------------*/

template <class T>
void CInstructionInc<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw CInstructionError("Invalid paramater count - must be 1");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
}

/*-------------------------------------------------------------------------*/

template <class T>
void CInstructionInc<T>::execute(CCPU<T> *cpu)
{
```

```
  assert(cpu != NULL);
  super::checkRegister(cpu, m_regidx1);
  cpu->getRegisters()[ m_regidx1 ]++;
}

/*============================================================================*/

/**
 * @class CInstructionDec
 *
 * Implementation of assembler command "dec"
 * Syntax: dec R1
 *   (R1--)
 */
template <class T>
class CInstructionDec
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionDec()
      : CInstruction<T>("dec")
    {}

    CInstructionDec *factory()
    {
      return new CInstructionDec;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionDec<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw CInstructionError("Invalid paramater count - must be 1");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionDec<T>::execute(CCPU<T> *cpu)
{
  assert(cpu != NULL);
  super::checkRegister(cpu, m_regidx1);
  cpu->getRegisters()[ m_regidx1 ]--;
}

/*============================================================================*/

/**
 * @class CInstructionAdd
 *
```

```
 * Implementation of assembler command "add"
 * Syntax: add R1, R2, R3
 *   (R1 = R2 + R3)
 */
template <class T>
class CInstructionAdd
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionAdd()
      : CInstruction<T>("add")
    {}

    CInstructionAdd *factory()
    {
      return new CInstructionAdd;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
    /** register number */
    unsigned m_regidx3;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionAdd<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 3)
    throw CInstructionError("Invalid paramater count - must be 3");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx2 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx3 = super::parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionAdd<T>::execute(CCPU<T> *cpu)
{
  assert(cpu != NULL);
  super::checkRegister(cpu, m_regidx1);
  super::checkRegister(cpu, m_regidx2);
  super::checkRegister(cpu, m_regidx3);
  cpu->getRegisters()[ m_regidx1 ] = cpu->getRegisters()[ m_regidx2 ]
    + cpu->getRegisters()[ m_regidx3 ];
}

/*============================================================================*/

/**
 * @class CInstructionSub
```

```
 *
 * Implementation of assembler command "sub"
 * Syntax: sub R1, R2, R3
 *   (R1 = R2 - R3)
 */
template <class T>
class CInstructionSub
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionSub()
      : CInstruction<T>("sub")
    {}

    CInstructionSub *factory()
    {
      return new CInstructionSub;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
    /** register number */
    unsigned m_regidx3;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionSub<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 3)
    throw CInstructionError("Invalid paramater count - must be 3");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx2 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx3 = super::parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionSub<T>::execute(CCPU<T> *cpu)
{
  assert(cpu != NULL);
  super::checkRegister(cpu, m_regidx1);
  super::checkRegister(cpu, m_regidx2);
  super::checkRegister(cpu, m_regidx3);
  cpu->getRegisters()[ m_regidx1 ] = cpu->getRegisters()[ m_regidx2 ]
    - cpu->getRegisters()[ m_regidx3 ];
}

/*============================================================================*/

/**
```

```
 * @class CInstructionMul
 *
 * Implementation of assembler command "mul"
 * Syntax: mul R1, R2, R3
 *   (R1 = R2 * R3)
 */
template <class T>
class CInstructionMul
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionMul()
      : CInstruction<T>("mul")
    {}

    CInstructionMul *factory()
    {
      return new CInstructionMul;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
    /** register number */
    unsigned m_regidx3;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionMul<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 3)
    throw CInstructionError("Invalid paramater count - must be 3");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx2 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx3 = super::parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionMul<T>::execute(CCPU<T> *cpu)
{
  super::checkRegister(cpu, m_regidx1);
  super::checkRegister(cpu, m_regidx2);
  super::checkRegister(cpu, m_regidx3);
  cpu->getRegisters()[ m_regidx1 ] = cpu->getRegisters()[ m_regidx2 ]
    * cpu->getRegisters()[ m_regidx3 ];
}

/*============================================================================*/

/**
```

```
 * @class CInstructionDiv
 *
 * Implementation of assembler command "div"
 * Syntax: div R1, R2, R3
 *   (R1 = R2 / R3)
 */
template <class T>
class CInstructionDiv
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionDiv()
      : CInstruction<T>("div")
    {}

    CInstructionDiv *factory()
    {
      return new CInstructionDiv;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
    /** register number */
    unsigned m_regidx3;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionDiv<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 3)
    throw CInstructionError("Invalid paramater count - must be 3");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx2 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx3 = super::parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionDiv<T>::execute(CCPU<T> *cpu)
{
  assert(cpu != NULL);
  super::checkRegister(cpu, m_regidx1);
  super::checkRegister(cpu, m_regidx2);
  super::checkRegister(cpu, m_regidx3);
  cpu->getRegisters()[ m_regidx1 ] = cpu->getRegisters()[ m_regidx2 ]
    / cpu->getRegisters()[ m_regidx3 ];
}

/*============================================================================*/
```

```
/**
 * @class CInstructionLoad
 *
 * Implementation of assembler command "load"
 * Syntax: load R1, R2
 *   (R1 = memory[R2])
 */
template <class T>
class CInstructionLoad
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionLoad()
      : CInstruction<T>("load")
    {}

    CInstructionLoad *factory()
    {
      return new CInstructionLoad;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
};

/*--------------------------------------------------------------------------*/

template <class T>
void CInstructionLoad<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 2)
    throw CInstructionError("Invalid paramater count - must be 2");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx2 = super::parseRegister(params.front());
  params.pop_front();
}

/*--------------------------------------------------------------------------*/

template <class T>
void CInstructionLoad<T>::execute(CCPU<T> *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getMemory() != NULL);
  super::checkRegister(cpu, m_regidx1);
  super::checkRegister(cpu, m_regidx2);
  T val(cpu->getRegisters()[ m_regidx2 ]);
  cpu->getRegisters()[ m_regidx1 ] = (*cpu->getMemory())[ val ];
}

/*==========================================================================*/

/**
 * @class CInstructionStore
 *
```

```
 * Implementation of assembler command "store"
 * Syntax: store R1, R2
 *   (memory[R2] = R1)
 */
template <class T>
class CInstructionStore
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionStore()
      : CInstruction<T>("store")
    {}

    CInstructionStore *factory()
    {
      return new CInstructionStore;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** register number */
    unsigned m_regidx2;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionStore<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 2)
    throw CInstructionError("Invalid paramater count - must be 2");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
  m_regidx2 = super::parseRegister(params.front());
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionStore<T>::execute(CCPU<T> *cpu)
{
  assert(cpu != NULL);
  assert(cpu->getMemory() != NULL);
  super::checkRegister(cpu, m_regidx1);
  super::checkRegister(cpu, m_regidx2);
  T val(cpu->getRegisters()[ m_regidx2 ]);
  (*cpu->getMemory())[ val ] = cpu->getRegisters()[ m_regidx1 ];
}

/*============================================================================*/

/**
 * @class CInstructionTest
 *
 * Implementation of assembler command "test"
 * Syntax: test R1
 *   (R1 == 0: zeroflag: true, R1 < 0: signflag: true)
```

```
 */
template <class T>
class CInstructionTest
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionTest()
      : CInstruction<T>("test")
    {}

    CInstructionTest *factory()
    {
      return new CInstructionTest;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
};

/*-------------------------------------------------------------------------*/

template <class T>
void CInstructionTest<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw CInstructionError("Invalid paramater count - must be 1");
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
}

/*-------------------------------------------------------------------------*/

template <class T>
void CInstructionTest<T>::execute(CCPU<T> *cpu)
{
  assert(cpu != NULL);
  super::checkRegister(cpu, m_regidx1);
  if (cpu->getRegisters()[ m_regidx1 ] == T(0))
    cpu->setFlagZero(true);
  if (cpu->getRegisters()[ m_regidx1 ] < T(0))
    cpu->setFlagSign(true);
}

/*=========================================================================*/

/**
 * @class CInstructionLabel
 *
 * Implementation of assembler command "label"
 * Syntax: label name:
 */
template <class T>
class CInstructionLabel
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
```

```
    CInstructionLabel()
      : CInstruction<T>("label")
    {}

    CInstructionLabel *factory()
    {
      return new CInstructionLabel;
    }

    void compile(std::list<std::string>& params)
    {}

    void execute(CCPU<T> *cpu)
    {}
};

/*============================================================================*/

/**
 * @class CInstructionJumpA
 *
 * Implementation of assembler command "jumpa"
 * Syntax: jumpa labelname
 *   (jump to labelname)
 */
template <class T>
class CInstructionJumpA
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionJumpA()
      : CInstruction<T>("jumpa"), m_addr("")
    {}

    CInstructionJumpA *factory()
    {
      return new CInstructionJumpA;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** labelname */
    std::string m_addr;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionJumpA<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw CInstructionError("Invalid paramater count - must be 1");
  m_addr = params.front();
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionJumpA<T>::execute(CCPU<T> *cpu)
```

```
{
  assert(cpu != NULL);
  assert(cpu->getProgram() != NULL);
  if (m_addr.empty())
    throw CInstructionError("Empty address");
  try
  {
    cpu->getRegisters()[ 0 ] = cpu->getProgram()->findLabel(m_addr);
  }
  catch(CProgramError& ex)
  {
    throw CInstructionError(ex.what());
  }
}

/*============================================================================*/

/**
 * @class CInstructionJumpZ
 *
 * Implementation of assembler command "jumpz"
 * Syntax: jumpz labelname
 *   (jump to labelname if zeroflag)
 */
template <class T>
class CInstructionJumpZ
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionJumpZ()
      : CInstruction<T>("jumpz"), m_addr("")
    {}

    CInstructionJumpZ *factory()
    {
      return new CInstructionJumpZ;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** labelname */
    std::string m_addr;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionJumpZ<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw CInstructionError("Invalid paramater count - must be 1");
  m_addr = params.front();
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionJumpZ<T>::execute(CCPU<T> *cpu)
{
```

```
  assert(cpu != NULL);
  assert(cpu->getProgram() != NULL);
  if (!cpu->getFlagZero())
    return;
  if (m_addr.empty())
    throw CInstructionError("Empty address");
  try
  {
    cpu->getRegisters()[ 0 ] = cpu->getProgram()->findLabel(m_addr);
  }
  catch(CProgramError& ex)
  {
    throw CInstructionError(ex.what());
  }
}

/*============================================================================*/

/**
 * @class CInstructionJumpS
 *
 * Implementation of assembler command "jumps"
 * Syntax: jumps labelname
 *   (jump to labelname if signflag)
 */
template <class T>
class CInstructionJumpS
 : public CInstruction<T>
{
  typedef CInstruction<T> super;

  public:
    CInstructionJumpS()
      : CInstruction<T>("jumps"), m_addr("")
    {}

    CInstructionJumpS *factory()
    {
      return new CInstructionJumpS;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** labelname */
    std::string m_addr;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionJumpS<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 1)
    throw CInstructionError("Invalid paramater count - must be 1");
  m_addr = params.front();
  params.pop_front();
}

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionJumpS<T>::execute(CCPU<T> *cpu)
```

```
{
  assert(cpu != NULL);
  assert(cpu->getProgram() != NULL);
  if (!cpu->getFlagSign())
    return;
  if (m_addr.empty())
    throw CInstructionError("Empty address");
  try
  {
    cpu->getRegisters()[ 0 ] = cpu->getProgram()->findLabel(m_addr);
  }
  catch(CProgramError& ex)
  {
    throw CInstructionError(ex.what());
  }
}

/*============================================================================*/

/**
 * @class CInstructionWrite
 *
 * Implementation of assembler command "write"
 * Syntax: write DEV, R1
 *   (write R1 to DEV, which is a name of a display)
 */
template <class T>
class CInstructionWrite
 : public CInstruction<T>
{
  typedef CInstruction<T> super;
  typedef typename std::set<CDisplay<T> *>::iterator setiterator;

  public:
    CInstructionWrite()
      : CInstruction<T>("write"), m_dev("")
    {}

    CInstructionWrite *factory()
    {
      return new CInstructionWrite;
    }

    void compile(std::list<std::string>& params);
    void execute(CCPU<T> *cpu);

  protected:
    /** register number */
    unsigned m_regidx1;
    /** device name */
    std::string m_dev;
};

/*----------------------------------------------------------------------------*/

template <class T>
void CInstructionWrite<T>::compile(std::list<std::string>& params)
{
  if (params.size() != 2)
    throw CInstructionError("Invalid paramater count - must be 2");
  m_dev = params.front();
  params.pop_front();
  m_regidx1 = super::parseRegister(params.front());
  params.pop_front();
```

```
}

/*-------------------------------------------------------------------------*/

template <class T>
void CInstructionWrite<T>::execute(CCPU<T> *cpu)
{
  assert(cpu != NULL);
  super::checkRegister(cpu, m_regidx1);
  if (m_dev.empty())
    throw CInstructionError("Empty device");

  CDisplay<T> *display = NULL;
  std::set<CDisplay<T> *> displays = cpu->getDisplays();
  for(setiterator it = displays.begin(); it != displays.end(); ++it)
  {
    if ((*it)->getName() == m_dev)
    {
      display = *it;
      break;
    }
  }
  if (display == NULL)
    throw CInstructionError("Unknown display");

  display->display(cpu->getRegisters()[ m_regidx1 ]);
}

#endif

/* vim: set et sw=2 ts=2: */
```

4.8 cdisplay.h

```
/**
 * @module cdisplay
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Abstract template class for displays
 * @date   26.05.2009
 */

#ifndef CDISPLAY_H
#define CDISPLAY_H 1

/**
 * @class CDisplay
 *
 * Abstract template class for displays
 */
template <class T>
class CDisplay
{
  public:
    /**
     * @method CDisplay
     * @brief  Default ctor
     * @param  name  name of display
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CDisplay(std::string name)
      : m_name(name)
    {}

    /**
     * @method ~CDisplay
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    virtual ~CDisplay()
    {}

    /**
     * @method getName
     * @brief  returns name of display
     * @param  -
     * @return name of display
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    virtual const std::string& getName()
    {
      return m_name;
    }
```

```
    /**
     * @method display
     * @brief  prints value to display
     * @param  value  value to display
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    virtual void display(const T &value) = 0;

  protected:
    /* members */
    /** name of display */
    std::string m_name;
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.9 displays.h

```
/**
 * @module displays
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  Implementations of CDisplay
 * @date   26.05.2009
 */

#ifndef DISPLAYS_H
#define DISPLAYS_H 1

#include <iomanip>
#include "cdisplay.h"

/**
 * @class CDisplayWDEZ
 *
 * Implementation of CDisplay
 * Prints T to stdout as decimal
 */
template <class T>
class CDisplayWDEZ
  : public CDisplay<T>
{
  public:
    CDisplayWDEZ()
      : CDisplay<T>("wdez")
    {}

    /**
     * @method display
     * @brief  prints value to display
     * @param  value  value to display
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    void display(const T &value)
    {
      std::cout << std::dec << value << std::endl;
    }
};

/*============================================================================*/

/**
 * @class CDisplayWHEX
 *
 * Implementation of CDisplay
 * Prints T to stdout as decimal
 */
template <class T>
class CDisplayWHEX
  : public CDisplay<T>
{
  public:
    CDisplayWHEX()
      : CDisplay<T>("whex")
    {}
```

```
    /**
     * @method display
     * @brief  prints value to display
     * @param  value  value to display
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    void display(const T &value)
    {
      std::cout << std::hex << value << std::endl;
    }
};

#endif

/* vim: set et sw=2 ts=2: */
```

4.10 cprogram.h

```
/**
 * @module cprogram
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  CProgram extends std::vector and adds a method for parsing programfile
 * @date   26.05.2009
 */

#ifndef CPROGRAM_H
#define CPROGRAM_H 1

#include <vector>
#include <set>
#include <map>
#include <stdexcept>
#include <boost/algorithm/string.hpp>
#include <boost/algorithm/string/split.hpp>
#ifdef DEBUG
# include <iostream>
# include <iomanip>
#endif

/**
 * @class CProgramError
 *
 * Exception thrown by implemententations of CProgram
 */
class CProgramError
 : public std::invalid_argument
{
  public:
    /**
     * @method CProgramError
     * @brief  Default exception ctor
     * @param  what  message to pass along
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CProgramError(const std::string& what)
      : std::invalid_argument(what)
    {}
};

#include "cinstruction.h"
#include "instructions.h"

/* forward declare CInstruction */
template <class T>
class CInstruction;

/**
 * @class CProgram
 *
 * CProgram extends std::vector and adds a method for parsing
 * programfile. This adds instances of CInstruction to CProgram itself.
 */
template <class T>
class CProgram
  : public std::vector<CInstruction<T> *>
```

```
{
  typedef typename std::set<CInstruction<T> *>::iterator setiterator;
  typedef std::vector<CInstruction<T> *> super;
  typedef typename super::iterator iterator;
  using super::begin;
  using super::end;
  using super::size;

  public:
    /**
     * @method CProgram
     * @brief  Default ctor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CProgram();

    /**
     * @method ~CProgram
     * @brief  Default dtor
     * @param  -
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    ~CProgram();

    /**
     * @method getLabels
     * @brief  get reference to labels map
     * @param  -
     * @return reference to labels map
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    const std::map<std::string, unsigned>& getLabels() const
    {
      return m_labels;
    }

    /**
     * @method findLabel
     * @brief  search for label
     * @param  label  name of label to search for
     * @return index of found label in program
     * @globalvars none
     * @exception  CProgramError
     * @pre  none
     * @post none
     */
    unsigned findLabel(const std::string& label) const;

    /**
     * @method compile
     * @brief  create instructions from parsing stream
     * @param  in  inputstream to read from
```

```
     * @return void
     * @globalvars none
     * @exception CProgramError
     * @pre none
     * @post none
     */
    void compile(std::istream& in);

#if DEBUG
    /**
     * @method dump
     * @brief dumps contents to outputstream
     * @param out outputstream to write to
     * @return void
     * @globalvars none
     * @exception none
     * @pre none
     * @post none
     */
    void dump(std::ostream& out);
#endif

  private:
    /* members */
    /** set of known instructions */
    std::set<CInstruction<T> *> m_instrset;
    std::map<std::string, unsigned> m_labels;
};

/*----------------------------------------------------------------------------*/

template <class T>
CProgram<T>::CProgram()
{
  m_instrset.insert(new CInstructionInc<T>);
  m_instrset.insert(new CInstructionDec<T>);
  m_instrset.insert(new CInstructionAdd<T>);
  m_instrset.insert(new CInstructionSub<T>);
  m_instrset.insert(new CInstructionMul<T>);
  m_instrset.insert(new CInstructionDiv<T>);
  m_instrset.insert(new CInstructionLoad<T>);
  m_instrset.insert(new CInstructionStore<T>);
  m_instrset.insert(new CInstructionTest<T>);
  m_instrset.insert(new CInstructionLabel<T>);
  m_instrset.insert(new CInstructionJumpA<T>);
  m_instrset.insert(new CInstructionJumpZ<T>);
  m_instrset.insert(new CInstructionJumpS<T>);
  m_instrset.insert(new CInstructionWrite<T>);
}

/*----------------------------------------------------------------------------*/

template <class T>
CProgram<T>::~CProgram()
{
  /* free instruction set */
  for (setiterator it = m_instrset.begin(); it != m_instrset.end(); ++it)
    delete *it;

  /* free instruction */
  for (iterator it2 = begin(); it2 != end(); ++it2)
    delete *it2;
}
```

```
/*-----------------------------------------------------------------------------*/

template <class T>
void CProgram<T>::compile(std::istream& in)
{
  if (!in.good())
    return;

  std::string line;
  unsigned i = 0;
  while (!in.eof() && in.good())
  {
    ++i;

    /* read stream per line */
    std::getline(in, line);
    if (line.empty())
      continue;

    boost::trim(line);
    boost::to_lower(line);

    /* ignore comments */
    if (line.find_first_of('#') == 0)
      continue;

    /* get instruction name */
    size_t pos = line.find_first_of(' ');
    std::string instrname(line.substr(0, pos));

    /* search and create instruction */
    CInstruction<T> *instrptr = NULL;
    setiterator it;
    for (it = m_instrset.begin(); it != m_instrset.end(); ++it)
    {
      if (*(*it) == instrname)
      {
        instrptr = *it;
        break;
      }
    }
    if (instrptr == NULL)
    {
      std::stringstream sstr;
      sstr << "Unknown instruction '" << instrname << "' on line " << i << ".";
      throw CProgramError(sstr.str());
    }

    /* create instruction */
    CInstruction<T> *instr = instrptr->factory();

    /* parse instruction parameters */
    std::string params = (pos == std::string::npos) ? "" : line.substr(pos + 1);
    boost::trim(params);
    std::list<std::string> instrparams;
    boost::split(instrparams, params, boost::is_any_of(", \t"), boost::
        token_compress_on);

    /* let instruction parse the parameters. catch+throw exception */
    try
    {
      /* handle label instruction ourself, but still add a dummy instruction */
      if (instrname == "label")
      {
```

```
        if (instrparams.size() != 1)
          throw CInstructionError("Invalid paramater count - must be 1");
        std::string label(instrparams.front());
        if (label.length() < 2 || label[ label.length() - 1] != ':')
          throw CInstructionError("Label has invalid syntax");
        m_labels[ label.substr(0, label.length() - 1) ] = size();
      }
      instr->compile(instrparams);
    }
    catch(CInstructionError& ex)
    {
      std::stringstream sstr;
      sstr << "Unable to compile instruction '" << instrname
           << "' (line " << i << "): " << ex.what();
      throw CProgramError(sstr.str());
    }

    push_back(instr);
  }
}

/*----------------------------------------------------------------------------*/

template <class T>
unsigned CProgram<T>::findLabel(const std::string& label) const
{
  std::map<std::string, unsigned>::const_iterator it;
  it = m_labels.find(label);
  if (it == m_labels.end())
    throw CProgramError("Unknown label '" + label + "'");
  return it->second;
}

/*----------------------------------------------------------------------------*/

#if DEBUG
template <class T>
void CProgram<T>::dump(std::ostream& out)
{
  out << "[PROGRAM DUMP]" << std::endl;
  unsigned i = 0;
  for(iterator it = begin(); it != end(); ++it)
  {
    out << "[" << std::setw(4) << std::setfill('0') << i << "]  "
        << *(*it) << std::endl;
    ++i;
  }
}
#endif

#endif

/* vim: set et sw=2 ts=2: */
```

4.11 ccpu.h

```
/**
 * @module ccpu
 * @author Guenther Neuwirth (0626638), Manuel Mausz (0728348)
 * @brief  CPU implementation. Used as a container for memory and instructions.
 *         Implements a run method to execute the program (= the instructions).
 * @date   26.05.2009
 */

#ifndef CCPU_H
#define CCPU_H 1

#include <iostream>
#include <set>
#include <stdexcept>
#ifdef DEBUG
# include <iostream>
# include <iomanip>
#endif

/**
 * @class CCPUError
 *
 * Exception thrown by implemententations of CCPU
 */
class CCPUError
 : public std::invalid_argument
{
  public:
    /**
     * @method CCPUError
     * @brief  Default exception ctor
     * @param  what  message to pass along
     * @return -
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    CCPUError(const std::string& what)
      : std::invalid_argument(what)
    {}
};

#include "cmem.h"
#include "displays.h"
#include "cprogram.h"

/* forward declare CProgram */
template <class T>
class CProgram;

/**
 * @class CCPU
 *
 * CPU implementation. Used as a container for memory and instructions.
 * Implements a run method to execute the program (= the instructions).
 */
template <class T>
class CCPU
{
  typedef typename std::set<CDisplay<T> *>::iterator displayiterator;
```

```
public:
  /**
   * @method CCPU
   * @brief  Default ctor
   * @param  cnt       number of registers to allocate for this cpu
   * @param  datatype  reference instance of datatype to copy from
   * @return -
   * @globalvars none
   * @exception  none
   * @pre  none
   * @post none
   */
  CCPU(const unsigned cnt, T& datatype);

  /**
   * @method ~CCPU
   * @brief  Default dtor
   * @param  -
   * @return -
   * @globalvars none
   * @exception  none
   * @pre  none
   * @post none
   */
  ~CCPU();

  /**
   * @method getRegisterCount
   * @brief  get number of registers
   * @param  -
   * @return number of registers
   * @globalvars none
   * @exception  none
   * @pre  none
   * @post none
   */
  const unsigned getRegisterCount() const
  {
    return m_regcnt;
  }

  /**
   * @method getRegisters
   * @brief  get reference to registers vector
   * @param  -
   * @return reference to registers vector
   * @globalvars none
   * @exception  none
   * @pre  none
   * @post none
   */
  std::vector<T> &getRegisters()
  {
    return m_registers;
  }

  /**
   * @method setMemory
   * @brief  set memory of cpu
   * @param  memory  pointer to memory
   * @return -
   * @globalvars none
   * @exception  none
```

```
 * @pre  none
 * @post none
 */
void setMemory(CMem<T> *memory)
{
  m_memory = memory;
}

/**
 * @method getMemory
 * @brief  get pointer to memory
 * @param  -
 * @return pointer to memory
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
CMem<T> *getMemory() const
{
  return m_memory;
}

/**
 * @method setProgram
 * @brief  set program to execute
 * @param  program  pointer to program
 * @return -
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
void setProgram(const CProgram<T> *program)
{
  m_program = program;
}

/**
 * @method getProgram
 * @brief  get pointer to program
 * @param  -
 * @return pointer to program
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
const CProgram<T> *getProgram()
{
  return m_program;
}

/**
 * @method getDisplays
 * @brief  get set of pointers to displays
 * @param  -
 * @return reference to set of pointers to displays
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
const std::set<CDisplay<T> *>& getDisplays()
```

```
{
  return m_displays;
}

/**
 * @method setFlagZero
 * @brief  set zero flag
 * @param  value  new value of zero flag
 * @return -
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
void setFlagZero(const bool value)
{
  m_flagzero = value;
}

/**
 * @method getFlagZero
 * @brief  get value of zero flag
 * @param  -
 * @return value of zero flag
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
const bool getFlagZero()
{
  return m_flagzero;
}

/**
 * @method setFlagSign
 * @brief  set sign flag
 * @param  value  new value of sign flag
 * @return -
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
void setFlagSign(const bool value)
{
  m_flagsign = value;
}

/**
 * @method getFlagSign
 * @brief  get value of sign flag
 * @param  -
 * @return value of sign flag
 * @globalvars none
 * @exception  none
 * @pre  none
 * @post none
 */
const bool getFlagSign()
{
  return m_flagsign;
}
```

```
    /**
     * @method run
     * @brief  execute current program
     * @param  -
     * @return -
     * @globalvars none
     * @exception  CCPUError
     * @pre  none
     * @post none
     */
    void run();

#if DEBUG
    /**
     * @method dumpRegisters
     * @brief  dump content of registers to outputstream
     * @param  out  outputstream to write to
     * @return void
     * @globalvars none
     * @exception  none
     * @pre  none
     * @post none
     */
    void dumpRegisters(std::ostream& out);
#endif

  private:
    /* members */
    T m_datatype;
    std::vector<T> m_registers;
    unsigned m_regcnt;
    CMem<T> *m_memory;
    const CProgram<T> *m_program;
    std::set<CDisplay<T> *> m_displays;
    bool m_flagzero;
    bool m_flagsign;
};

/*----------------------------------------------------------------------------*/

template <class T>
CCPU<T>::CCPU(const unsigned cnt, T& datatype)
  : m_datatype(datatype), m_registers(cnt, T(m_datatype) = 0), m_regcnt(cnt),
      m_memory(NULL), m_program(NULL), m_flagzero(false), m_flagsign(false)
{
  /* create displays */
  m_displays.insert(new CDisplayWDEZ<T>);
  m_displays.insert(new CDisplayWHEX<T>);
}

/*----------------------------------------------------------------------------*/

template <class T>
CCPU<T>::~CCPU()
{
  /* delete displays */
  for (displayiterator it = m_displays.begin() ; it != m_displays.end(); ++it)
    delete *it;
}

/*----------------------------------------------------------------------------*/

template <class T>
void CCPU<T>::run()
```

```
{
  if (m_memory == NULL)
    throw CCPUError("CPU␣has␣no␣memory");
  if (m_program == NULL)
    throw CCPUError("CPU␣has␣no␣program␣to␣execute");
  if (m_regcnt == 0)
    throw CCPUError("CPU␣has␣no␣registers");

  bool run = true;
  while(run)
  {
    unsigned pc = static_cast<unsigned>(m_registers[0]);

    /* end of the program reached */
    if (pc == m_program->size())
      break;

    /* pc is out of bound */
    if (pc > m_program->size())
      throw CCPUError("Programcounter␣is␣out␣of␣bound");

    /* execute instruction */
    try
    {
      (*m_program->at(pc))(this);
      ++m_registers[0];
    }
    catch(CInstructionError& ex)
    {
      throw CCPUError(ex.what());
    }
  }
}

/*-----------------------------------------------------------------------------*/

#if DEBUG
template <class T>
void CCPU<T>::dumpRegisters(std::ostream& out)
{
  out << "[REGISTER␣DUMP]" << std::endl;
  for(unsigned i = 0; i < getRegisterCount(); ++i)
  {
    out << "[" << std::setw(4) << std::setfill('0') << i << "]␣␣"
        << m_registers[i] << std::endl;
  }
}
#endif

#endif

/* vim: set et sw=2 ts=2: */
```