

Rechnernetze und Organisation

Framework für Assignment A1

Übersicht

Arbeitsumgebung

- Linux und GCC

Framework für Assignment A1

- Makefile-Umgebung
- Klassen
- Addition in C/C++
- Addition in Inline-Assembler

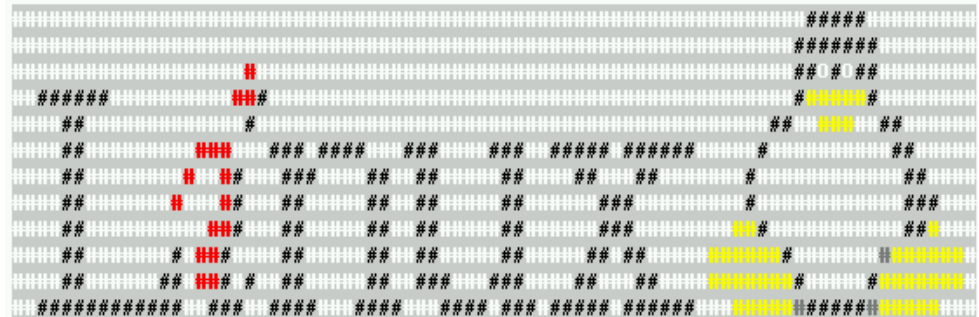
Arbeitsumgebung

Linux und GCC

- Pluto als Referenz-Plattform
 - Vor Abgabe dort testen

Möglichkeiten

- Subzentren: Linux
- Linux installieren
- Linux unter Windows
 - VM-Ware Image: Debian
 - Cygwin
 - MinGW
- SSH auf Pluto.tugraz.at
 - X11-Server am PC
 - Cygwin, Exceed
 - SSH-Verbindung



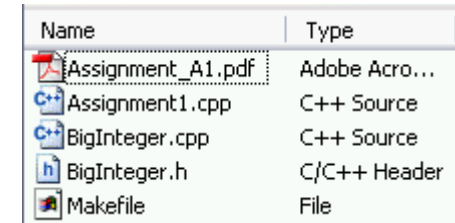
Framework Aufbau

RNO_A1.zip

- Download von RNO-Webseiten
- Enthält: Makefile, Assignment1.cpp, BigInteger.cpp, BigInteger.h, Dokumentation
- Entpacken: > *unzip RNO_A1.zip*

Dateien

- BigInteger.cpp, BigInteger.h
 - Implementation der Klasse BigInteger
- Assignment1.cpp
 - Enthält main()
 - Enthält Testdaten
 - Demonstriert Funktionalität der BigInteger-Klasse
- Makefile



Name	Type
Assignment_A1.pdf	Adobe Acro...
Assignment1.cpp	C++ Source
BigInteger.cpp	C++ Source
BigInteger.h	C/C++ Header
Makefile	File

Framework für Assignment A1

Makefile-Umgebung

- Zum Compilieren
 - > *make compile*
- Zum Ausführen und Debuggen
 - > *make run*
 - > *make dbg*
- Zum Zusammenpacken der Abgabefiles
 - > *make dist*

```

TITLE = Assignment_A1

OBJS := $(patsubst %.cpp,%.o,$(wildcard *.cpp))
CC = g++
LD = g++
CC_FLAGS = -c -g -m32
EXECUTABLE = $(TITLE)
LD_FLAGS = -m32 -o $(EXECUTABLE)

$(EXECUTABLE) : $(OBJS)
    $(LD) $(LD_FLAGS) $(OBJS)

%.o: %.cpp *.h
    $(CC) $(CC_FLAGS) $<

$(TITLE).zip : *.h *.cpp Makefile
    zip $@ $^

all : $(EXECUTABLE)

run : $(EXECUTABLE)
    ./$^

dbg : $(EXECUTABLE)
    ddd ./$^

dist : $(TITLE).zip
    @echo
    @echo "Online-Abgabe von '$<':"
```

Makefiles

- Erzeugen Targets (abhängige Files)
 - Wenn sich Dependencies (Source-Dateien) ändern

Bestehender Code

BigInteger-Klasse

- Einfach gehaltene Klasse für Langzahlenoperation
 - Kapselung von Daten und Code
 - BigInteger.h: Interface nach außen
 - BigInteger.cpp: Implementierung
 - Jede Langzahl ist Instanz der Klasse BigInteger

BigInteger.h (Ausschnitt)

```
class BigInteger
{
private:
    bigIntType value[BIG_INTEGER_MAX_WORDS];
public:
    BigInteger(void);
    BigInteger(bigIntType val);
    BigInteger(const char str[]);
    ~BigInteger(void);
    BigInteger& addc(const BigInteger &addend);
    BigInteger& adda(const BigInteger &addend);
    BigInteger& shiftLeft(unsigned int val);
    BigInteger& modp192c();
    BigInteger& modp192a();
    bool compare(const BigInteger &testme);
};
```

BigInteger.cpp (Ausschnitt)

```
BigInteger::BigInteger(void)
{
    // Initialize the BigInteger with value to 0
    for (int i=0; i<BIG_INTEGER_MAX_WORDS; i++)
        value[i] = 0;
}
```

Verwenden von BigInteger

Anlegen von BigInteger Zahlen

- Deklaration
 - *BigInteger varname;*

- Definition
 - *varname = BigInteger(arg)*
 - Durch Aufruf von Constructor
 - Verschiedene Constructoren
 - » *BigInteger()*
 - » *BigInteger(long val)*
 - » *BigInteger(char hexadecimal_string[])*

- Framework legt BigIntegers nur statisch an
 - Wegen Einfachheit: Kein dynamisches Speichermanagement nötig

Assignment1.cpp (Ausschnitt)

```
void printConstants()
{
    BigInteger big12345678(0x12345678);
    cout << "big12345678 = " << big12345678 << endl;
    BigInteger big_string("000123456789abcDEF");
    cout << "big_string = " << big_string << endl;
}
```

Verwenden von BigInteger

Rechnen mit BigInteger

– Addition ist bereits implementiert

- addc()
- adda()

BigInteger.cpp (Ausschnitt)

```
BigInteger& BigInteger::addc(const BigInteger &addend)
{
    // Multi-precision addition: carry handling is impo
    bigIntType carry_next, carry = 0;
    for (int i=0; i<BIG_INTEGER_MAX_WORDS; i++) {
        carry_next = addend.value[i];
        value[i] += addend.value[i];
        carry_next = value[i] < carry_next;
        value[i] += carry;
        carry = carry_next || (value[i] < carry);
    }
    return *this;
}
```

Assignment1.cpp (Ausschnitt)

```
BigInteger big12345678(0x12345678);
cout << "0x12345678 + 0x000000001 = " << big12345678.adda(BIG1) << endl;

BigInteger big_msb(0x80000000);
cout << "0x80000000 + 0x80000000 = " << big_msb.adda(big_msb) << endl;

BigInteger big_max(0xFFFFFFFF);
cout << "0xFFFFFFFF + 0x000000001 = " << big_max.adda(BIG1) << endl;
```


Implementierungsdetails

Datentypen: Wie werden Langzahlen gespeichert?

- Unterteilung in 12 32-Bit Wörter
- Als Array von Longs
 - $value[i]$ $i = 0 \dots 11$

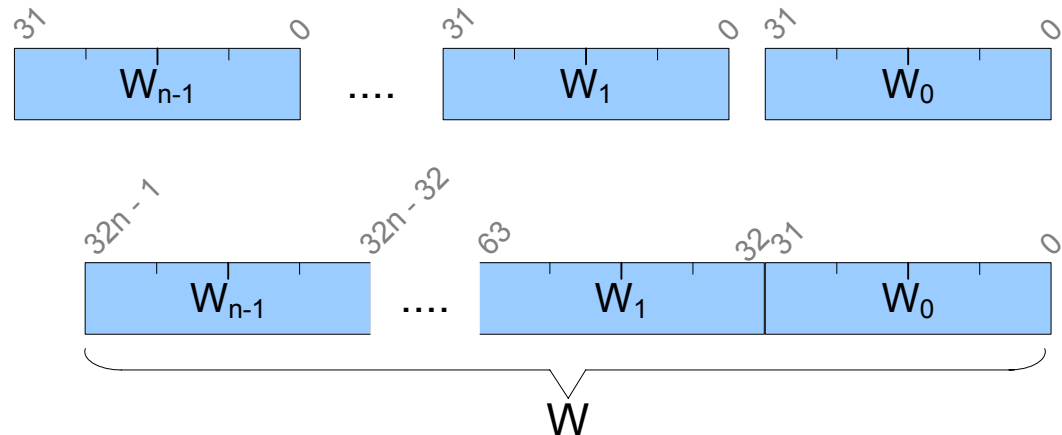
BigInteger.h (Ausschnitt)

```
// type definition of word-level datatype
typedef unsigned long bigIntType;

// Number of 32-bit words concatenated to BigInteger
#define BIG_INTEGER_MAX_WORDS 12

class BigInteger
{
private:
    bigIntType value[BIG_INTEGER_MAX_WORDS];
public:
    ...
}
```

$$W = \sum_{i=0}^{n-1} W_i 2^{32i}$$



Implementierungsdetails

Addition in C/C++

- Handling von Carries
 - Übertrag muss in nächste 32-Bit Stelle gelangen

BigInteger.cpp (Ausschnitt)

```
BigInteger& BigInteger::add(const BigInteger &addend)
{
    // Multi-precision addition: carry handling is impc
    bigIntType carry_next, carry = 0;
    for (int i=0; i<BIG_INTEGER_MAX_WORDS; i++) {
        carry_next = addend.value[i];
        value[i] += addend.value[i];
        carry_next = value[i] < carry_next;
        value[i] += carry;
        carry = carry_next || (value[i] < carry);
    }
    return *this;
}
```

- Feststellen ob Carry auftritt:
 - Ergebnis von 32-Bit Addition ist kleiner als Addend
 - » `carry_next = (value[i] < carry_next)`
- Berücksichtigen von früheren Carries
 - `value[i] += carry;`
 - Kann wiederum Carry produzieren
 - » `(value[i] < carry)`

Implementierungsdetails

Addition in Assembler

BigInteger.cpp (Ausschnitt)

```

BigInteger& BigInteger::adda(const BigInteger &addend)
{  // Multi-precision addition: carry handling is important

    __asm__ __volatile__(
        "MOV  %0,  %%esi\n\t"           // esi -> addend
        "MOV  %1,  %%edi\n\t"           // edi -> destination
        "MOV  $12, %%ecx\n\t"           // number of iterations
        "XOR  %%edx,%%edx\n\t"           // start with index 0
        "addition_loop:\n\t"
        "MOVL (%%esi, %%edx,4), %%eax\n\t" // EAX = addend.value[edx]
        "ADCL (%%edi, %%edx,4), %%eax\n\t" // EAX += value[edx] + carry
        "MOVL %%eax, (%%edi,%%edx,4)\n\t" // value[edx] = EAX
        "INC  %%edx\n\t"
        "LOOP addition_loop\n\t"         // decrement ECX and loop
        :                               // output variables
        : "a"(&addend), "c"(this)       // input variables
        : "%esi", "%edi", "%edx", "memory" // clobber stuff
    );

    return *this;
}

```

- Carry wird durch „Add with Carry (ADC) berücksichtigt“

Wie starten?

1. Algorithmus (Modulare Reduktion mod p192) analysieren

- Am Papier durchexerzieren
- Fortschritt im Tagesjournal konsequent protokollieren

2. C-Algorithmus erstellen

- Leere Funktionen ausfüllen

```
-----  
/*  
 * Modular reduction  
 */  
BigInteger& BigInteger::modp192c()  
{ // Modular reduction mod p192 = 2^192 - 2^64 -1  
  
    // add your c code of the reduction algorithm here  
    return *this;  
}  
  
BigInteger& BigInteger::modp192a()  
{ // Modular reduction mod p192 = 2^192 - 2^64 -1  
  
    // add your inline-assembler code of the algorithm here  
    return *this;  
}
```

3. Algorithmus testen

- Erstellen einfacher Testfälle
- Erstellen “großer” Testfälle

4. Assembler-Algorithmus erstellen und testen

- Wie viele Variablen werden benötigt?
- Welche Ressourcen stehen zur Verfügung (z.B. Register)?
- Implementierung des Algorithmus
- Dokumentation jeder Zeile