Hardwarenahe Programmierung I

U. Kaiser, R. Kaiser, M. Stöttinger, S. Reith

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

Gebraucht der Zeit, sie geht so schnell von hinnen, doch Ordnung lehrt euch Zeit gewinnen.

Johann Wolfgang von Goethe

http://www.denkschatz.de/zitate/Johann-Wolfgang-von-Goethe/Gebraucht-der-Zeit-sie-geht-so-schnell-von-hinnen-doch-Ordnung-lehrt-euch-Zeit/

Der Preprocessor

- Anweisungen, die am Zeilenanfang mit # beginnen, richten sich an den **Preprocessor**.
- Der Preprocessor verarbeitet diese Anweisungen bevor der Compiler mit der Übersetzung des Programms beginnt. Der Compiler übersetzt dann den durch den Preprocessor vorverarbeiteten Quellcode.

```
# include <stdio.h>
# include <stdlib.h>


void main()
{
    ...
    ...
    ...
}
```

- Der Preprocessor erzeugt keinen ausführbaren Code sondern führt nur Textersetzungen im Quellcode durch.

Der Preprocessor

Hochschule RheinMain

- Anweisungen, die am Zeilenanfang mit # beginnen, richten sich an den **Preprocessor**.
- Der Preprocessor verarbeitet diese Anweisungen bevor der Compiler mit der Übersetzung des Programms beginnt. Der Compiler übersetzt dann den durch den Preprocessor vorverarbeiteten Quellcode.

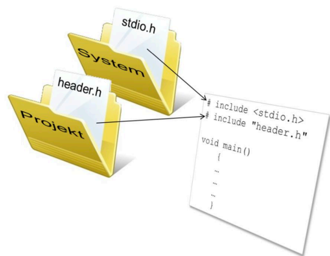
```
# include <stdio.h>
# include <stdlib.h>


void main()
{
    ...
    ...
    ...
}
```

Um diese Anweisungen geht es unter anderem in diesem Kapitel.

- Der Preprocessor erzeugt keinen ausführbaren Code sondern führt nur Textersetzungen im Quellcode durch.

Include Anweisung

- Mit einer `#include`-Anweisung können komplette Dateien vor der Übersetzung virtuell in den Quellcode eingefügt („includiert“) werden. Üblicherweise handelt es sich dabei sogenannte *Header*-Dateien. Diese Dateien erkennt man an der Namenserweiterung "`.h`".
- **System-Headerdateien** sind Dateien, die mit dem Compiler oder mit speziellen System- oder Entwicklungskomponenten geliefert werden und auf dem Entwicklungsrechner bereits vorhanden sind. Diese Dateien liegen in speziellen Systemverzeichnissen, die der Entwicklungsumgebung bekannt sind.
- **Projekt-Headerdateien** sind Headerdateien, die Sie in Ihrem Projekt selbst erstellen. Diese Dateien liegen zusammen mit den von Ihnen ebenfalls erstellten Quellcodedateien im Projektordner Ihres Projekts.



- In einer `#include`-Anweisung werden
  - **System-Headerdateien** in spitze Klammern (`<...>`) und
  - **Projekt-Headerdateien** in Anführungszeichen (`"..."`) gesetzt.
- In Headerdateien stehen Informationen, die einer oder mehreren Quellcodedateien von zentraler Stelle aus einheitlich zur Verfügung gestellt werden sollen.

Arbeitsweise der Include Anweisung

Hochschule RheinMain

- Bei der Verarbeitung einer Include-Anweisung ersetzt der Preprocessor die Anweisung durch den kompletten Inhalt der angesprochenen Datei. Der Compiler liest den Quelltext so, als würden die includierten Dateien anstelle der Include-Anweisung im Text stehen:

```
// Quellcodedatei
...
...
# include "header1.h"
...
...
...
...
...
...
...
```

```
// header1.h
...
...
# include "header2.h"
...
...
...
```

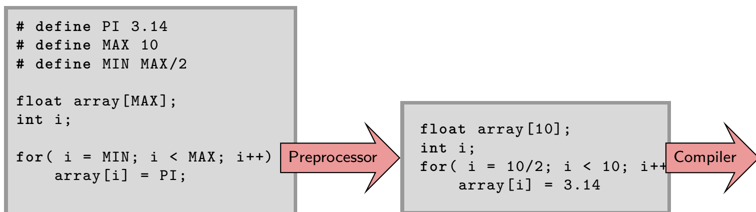
```
// header2.h
...
...
...
...
...
```

- Includierte Dateien können ihrerseits wieder Includes enthalten.

Symbolische Konstanten

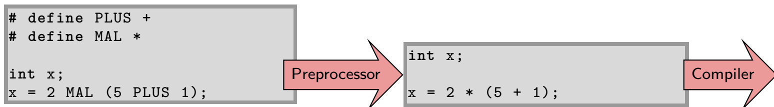
Hochschule RheinMain

- Durch **symbolische Konstanten können** Werte, die an unterschiedlichen Stellen im Quellcode einheitlich verwendet werden sollen, an zentraler Stelle festgelegt und gepflegt werden.



Symbolische Konstanten sind keine Variablen sondern nur Platzhalter für einen Ersatztext!

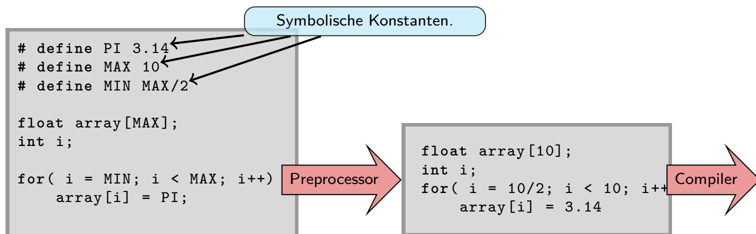
- Es können sehr allgemeine Ersetzungen durchgeführt werden. Wichtig ist, dass nach der Verarbeitung durch den Preprocessor gültiger Quellcode entsteht:



Symbolische Konstanten

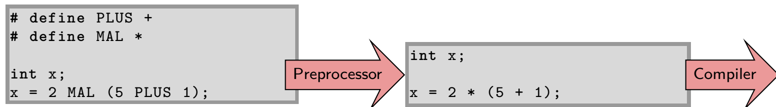
Hochschule RheinMain

- Durch **symbolische Konstanten können** Werte, die an unterschiedlichen Stellen im Quellcode einheitlich verwendet werden sollen, an zentraler Stelle festgelegt und gepflegt werden.



Symbolische Konstanten sind keine Variablen sondern nur Platzhalter für einen Ersatztext!

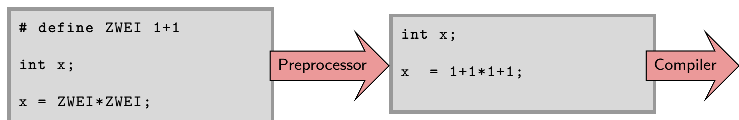
- Es können sehr allgemeine Ersetzungen durchgeführt werden. Wichtig ist, dass nach der Verarbeitung durch den Preprocessor gültiger Quellcode entsteht:



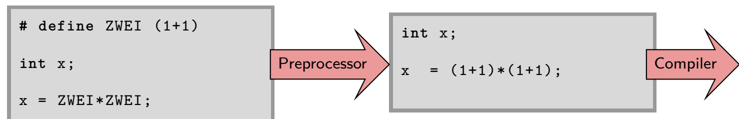
Auflösung symbolischer Konstanten

Hochschule RheinMain

- Bei der Auflösung von symbolischen Konstanten können unerwünschte Effekte auftreten:



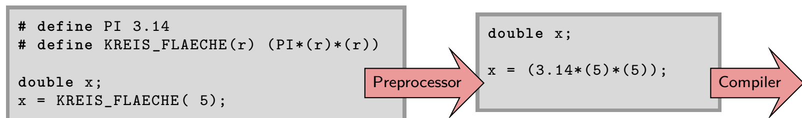
- Ausdrücke werden nicht ausgewertet, vereinfacht oder ausgerechnet. Es findet eine reine Textersetzung statt.
- Setzen Sie um Ausdrücke „Sicherheitsklammern“, da Sie nicht wissen, in welchem Kontext die Auflösung erfolgt:



Preprocessor-Makros

Hochschule RheinMain

- **Makros** ermöglichen es, Textersetzungen über Parameter zu steuern:



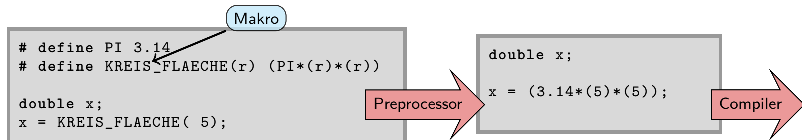
- Bei der Ersetzung durch den Preprocessor werden keine Auswertungen, Vereinfachungen oder Berechnungen durchgeführt, auch hier handelt es sich um eine reine Textersetzung.

Makros sind keine Funktionen, sondern nur parametrierte Platzhalter für einen Ersatztext!

Preprocessor-Makros

Hochschule RheinMain

- **Makros** ermöglichen es, Textersetzungen über Parameter zu steuern:



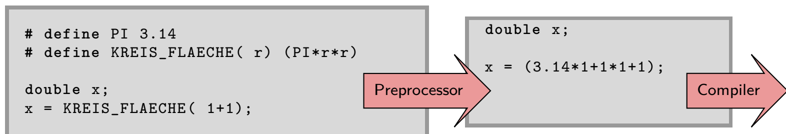
- Bei der Ersetzung durch den Preprocessor werden keine Auswertungen, Vereinfachungen oder Berechnungen durchgeführt, auch hier handelt es sich um eine reine Textersetzung.

Makros sind keine Funktionen, sondern nur parametrierte Platzhalter für einen Ersatztext!

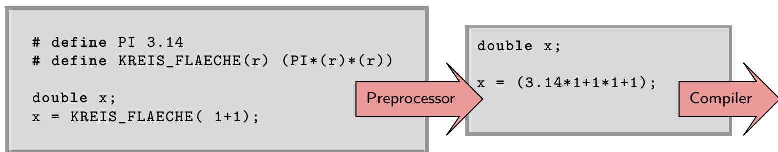
Auflösung von Makros

Hochschule RheinMain

- Bei der Auflösung von Makros kann es zu unerwünschten Effekten kommen:



- Setzen Sie Klammern um Parameter, um ungewollte Effekte zu vermeiden.



Auflösung von Makros

Hochschule RheinMain

- Achten Sie auf Seiteneffekte bei Formelausdrücken in Makros

```
# define PI 3.14
# define KREIS_FLAECHE(r) (PI*(r)*(r))

double x;
int a = 1;
x = KREIS_FLAECHE( a++);
```

Preprocessor

```
double x;

x = (3.14*(a++)*(a++));
```

Compiler

- Nach der Auflösung des Makros wird im letzten Beispiel a zweimal inkrementiert, was wahrscheinlich nicht beabsichtigt war.

Beispiel aus der Praxis

Hochschule RheinMain

- Implementierungsvorschlag[1] der NSA für den symmetrischen Blockverschlüsselungsalgorithmus Simon128/128
- Die Typen- und Basisoperationsdefinitionen des Verschlüsslungsalgorithmus sind als Macro implementiert

```
#define LCS _lrotl  //left circular shift
#define u64 unsigned long long
#define f(x) ((LCS(x,1) & LCS(x,8)) ^ LCS(x,2))
#define R2(x,y,k1,k2) (y^=f(x), y^=k1, x^=f(y), x^=k2)

void Simon128Encrypt(u64 pt[], u64 ct[], u64 k[])
{
    u64 i;

    ct[0]=pt[0]; ct[1]=pt[1];
    for(i=0; i<68; i+=2) R2(ct[1], ct[0], k[i], k[i+1]);
  }
```

[1] https://eprint.iacr.org/2013/404.pdf

Beispiel aus der Praxis

Hochschule RheinMain

- Implementierungsvorschlag[1] der NSA für den symmetrischen Blockverschlüsselungsalgorithmus `Simon128/128`
- Die Typen- und Basisoperationsdefinitionen des Verschlüsslungsalgorithmus sind als Macro implementiert

Definition einer Linksrotationsfunktion

```
#define LCS _lrotl  //left circular shift
#define u64 unsigned long long
#define f(x) ((LCS(x,1) & LCS(x,8)) ^ LCS(x,2))
#define R2(x,y,k1,k2) (y^=f(x), y^=k1, x^=f(y), x^=k2)

void Simon128Encrypt(u64 pt[], u64 ct[], u64 k[])
{
    u64 i;

    ct[0]=pt[0]; ct[1]=pt[1];
    for(i=0; i<68; i+=2) R2(ct[1], ct[0], k[i], k[i+1]);
  }
```

[1] https://eprint.iacr.org/2013/404.pdf

Beispiel aus der Praxis

Hochschule RheinMain

- Implementierungsvorschlag[1] der NSA für den symmetrischen Blockverschlüsselungsalgorithmus `Simon128/128`
- Die Typen- und Basisoperationsdefinitionen des Verschlüsslungsalgorithmus sind als Macro implementiert

Definition einer Linksrotationsfunktion

Definition eines vorzeichenfreien 64-Bit Wertes

```
#define LCS _lrotl  //left circular shift
#define u64 unsigned long long
#define f(x) ((LCS(x,1) & LCS(x,8)) ^ LCS(x,2))
#define R2(x,y,k1,k2) (y^=f(x), y^=k1, x^=f(y), x^=k2)

void Simon128Encrypt(u64 pt[], u64 ct[], u64 k[])
{
    u64 i;

    ct[0]=pt[0]; ct[1]=pt[1];
    for(i=0; i<68; i+=2) R2(ct[1], ct[0], k[i], k[i+1]);
  }
```

[1] https://eprint.iacr.org/2013/404.pdf

Beispiel aus der Praxis

Hochschule RheinMain

- Implementierungsvorschlag[1] der NSA für den symmetrischen Blockverschlüsselungsalgorithmus `Simon128/128`
- Die Typen- und Basisoperationsdefinitionen des Verschlüsslungsalgorithmus sind als Macro implementiert

```
#define LCS _lrotl  //left circular shift
#define u64 unsigned long long
#define f(x) ((LCS(x,1) & LCS(x,8)) ^ LCS(x,2))
#define R2(x,y,k1,k2) (y^=f(x), y^=k1, x^=f(y), x^=k2)

void Simon128Encrypt(u64 pt[], u64 ct[], u64 k[])
{
    u64 i;

    ct[0]=pt[0]; ct[1]=pt[1];
    for(i=0; i<68; i+=2) R2(ct[1], ct[0], k[i], k[i+1]);
  }
```

Definition einer Linksrotationsfunktion

Definition eines vorzeichenfreien 64-Bit Wertes

Basisoperationen

[1] https://eprint.iacr.org/2013/404.pdf

Beispiel aus der Praxis

Hochschule RheinMain

- Implementierungsvorschlag[1] der NSA für den symmetrischen Blockverschlüsselungsalgorithmus Simon128/128
- Die Typen- und Basisoperationsdefinitionen des Verschlüsslungsalgorithmus sind als Macro implementiert

```
#define LCS _lrotl  //left circular shift
#define u64 unsigned long long
#define f(x) ((LCS(x,1) & LCS(x,8)) ^ LCS(x,2))
#define R2(x,y,k1,k2) (y^=f(x), y^=k1, x^=f(y), x^=k2)

void Simon128Encrypt(u64 pt[], u64 ct[], u64 k[])
{
    u64 i;

    ct[0]=pt[0]; ct[1]=pt[1];
    for(i=0; i<68; i+=2) R2(ct[1], ct[0], k[i], k[i+1]);
  }
```

Definition einer Linksrotationsfunktion (→ #define LCS)

Definition eines vorzeichenfreien 64-Bit Wertes (→ #define u64)

Basisoperationen (→ #define f, #define R2)

Verschlüsselungsfunktion (→ Simon128Encrypt)

[1] https://eprint.iacr.org/2013/404.pdf

Beispiel aus der Praxis

Hochschule RheinMain

- Implementierungsvorschlag[1] der NSA für den symmetrischen Blockverschlüsselungsalgorithmus `Simon128/128`
- Die Typen- und Basisoperationsdefinitionen des Verschlüsslungsalgorithmus sind als Macro implementiert

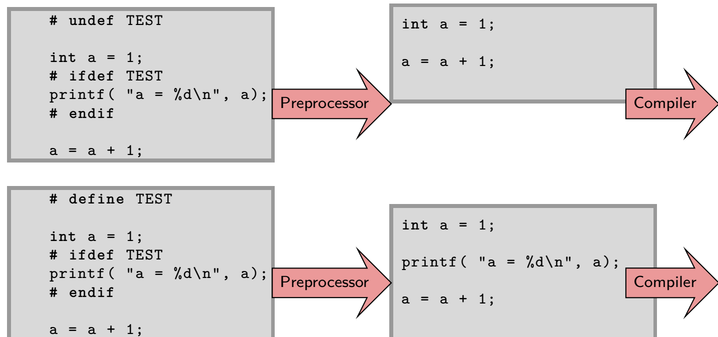
```
#define LCS _lrotl  //left circular shift
#define u64 unsigned long long
#define f(x) ((LCS(x,1) & LCS(x,8)) ^ LCS(x,2))
#define R2(x,y,k1,k2) (y^=f(x), y^=k1, x^=f(y), x^=k2)

void Simon128Encrypt(u64 pt[], u64 ct[], u64 k[])
{
    u64 i;

    ct[0]=pt[0]; ct[1]=pt[1];
    for(i=0; i<68; i+=2) R2(ct[1], ct[0], k[i], k[i+1]);
  }
```


[1] https://eprint.iacr.org/2013/404.pdf

Compileschalter

Hochschule RheinMain

- Durch **Compileschalter** können Teile des Codes von der Übersetzung ausgeschlossen werden oder verschiedene Varianten des Quellcodes aus einer Quelle erzeugt werden:

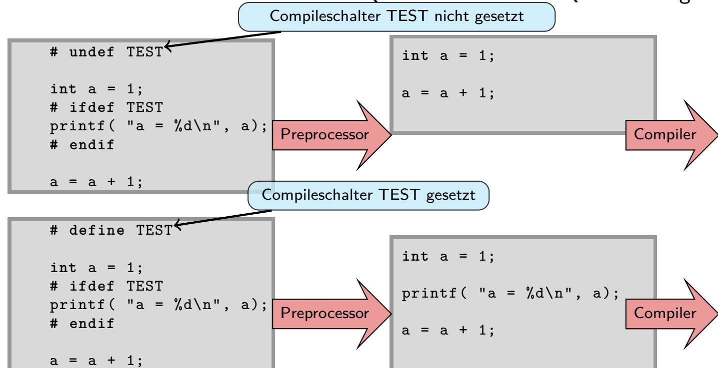


- Wenn der Compileschalter `TEST` gesetzt ist, sind zusätzlich Prüfdrucke im Code vorhanden. Ist der Compileschalter nicht gesetzt, sind die Prüfdrucke nicht vorhanden.

Compileschalter

Hochschule RheinMain

- Durch **Compileschalter** können Teile des Codes von der Übersetzung ausgeschlossen werden oder verschiedene Varianten des Quellcodes aus einer Quelle erzeugt werden:



- Wenn der Compileschalter `TEST` gesetzt ist, sind zusätzlich Prüfdrucke im Code vorhanden. Ist der Compileschalter nicht gesetzt, sind die Prüfdrucke nicht vorhanden.

Compileschalter

Hochschule **RheinMain**

- **Compileschalter sind keine if-Anweisungen**
  Eine if-Anweisung wird zur Laufzeit ausgeführt, ein Compileschalter wird durch den Preprocessor aufgelöst und ist zur Laufzeit nicht mehr im Code vorhanden.
- Compileschalter können im Quelltext wie sybmolische Konstanten definiert werden ...

```
# define TEST1 1
# undef TEST2
...
#ifdef TEST1
    printf("Dieser Text wird gedruckt\n");
#endif
#if TEST1==0
    printf("Dieser Text erscheint nicht!\n");
#endif
#ifdef TEST2
    printf("Dieser Text erscheint auch nicht!\n");
#endif
```

- ... auf der Compiler-Kommandozeile ...

```
.....
$ gcc -DTEST1=1 -UTEST2 -o hello hello.c
.....
```

- ... oder im `Makefile` (vgl. Kapitel 6)

```
.....
CFLAGS=-DTEST1=1 -UTEST2
hello: hello.c
.....
```

Compileschalter

Hochschule RheinMain

- **Compileschalter sind keine if-Anweisungen**
  Eine if-Anweisung wird zur Laufzeit ausgeführt, ein Compileschalter wird durch den Preprocessor aufgelöst und ist zur Laufzeit nicht mehr im Code vorhanden.
- Compileschalter können im Quelltext wie sybmolische Konstanten definiert werden ...

```
# define TEST1 1
# undef TEST2
...
#ifdef TEST1
    printf("Dieser Text wird gedruckt\n");
#endif
#if TEST1==0
    printf("Dieser Text erscheint nicht!\n");
#endif
#ifdef TEST2
    printf("Dieser Text erscheint auch nicht!\n");
#endif
```

TEST1 ist definiert

- ... auf der Compiler-Kommandozeile ...

```
.....
$ gcc -DTEST1=1 -UTEST2 -o hello hello.c
.....
```

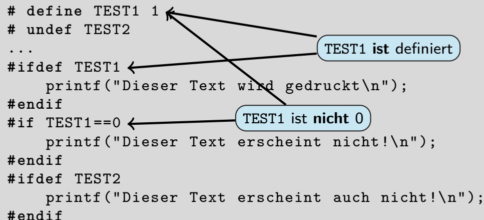
- ... oder im `Makefile` (vgl. Kapitel 6)

```
.....
CFLAGS=-DTEST1=1 -UTEST2
hello: hello.c
.....
```

Compileschalter

Hochschule RheinMain

- **Compileschalter sind keine if-Anweisungen**
  Eine if-Anweisung wird zur Laufzeit ausgeführt, ein Compileschalter wird durch den Preprocessor aufgelöst und ist zur Laufzeit nicht mehr im Code vorhanden.
- Compileschalter können im Quelltext wie sybmolische Konstanten definiert werden ...



```
# define TEST1 1
# undef TEST2
...
#ifdef TEST1
    printf("Dieser Text wird gedruckt\n");
#endif
#if TEST1==0
    printf("Dieser Text erscheint nicht!\n");
#endif
#ifdef TEST2
    printf("Dieser Text erscheint auch nicht!\n");
#endif
```


- ... auf der Compiler-Kommandozeile ...

```
.....
$ gcc -DTEST1=1 -UTEST2 -o hello hello.c
.....
```

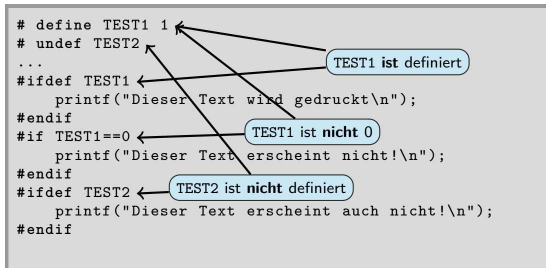
- ... oder im `Makefile` (vgl. Kapitel 6)

```
.....
CFLAGS=-DTEST1=1 -UTEST2
hello: hello.c
.....
```

Compileschalter

Hochschule RheinMain

- **Compileschalter sind keine if-Anweisungen**
  Eine if-Anweisung wird zur Laufzeit ausgeführt, ein Compileschalter wird durch den Preprocessor aufgelöst und ist zur Laufzeit nicht mehr im Code vorhanden.
- Compileschalter können im Quelltext wie sybmolische Konstanten definiert werden ...



- ... auf der Compiler-Kommandozeile ...

```
.....
$ gcc -DTEST1=1 -UTEST2 -o hello hello.c
.....
```

- ... oder im `Makefile` (vgl. Kapitel 6)

```
.....
CFLAGS=-DTEST1=1 -UTEST2
hello: hello.c
.....
```

„Eingebaute“ Compileschalter

Hochschule RheinMain

- Compiler definieren in der Regel eine (große) Anzahl von Compileschaltern vor, aus denen auf verschiedene Eigenschaften des verwendeten Compilers geschlossen werden kann.
- Damit können im Code Plattformabhängigkeiten automatisch angepasst werden.

```
#ifdef __amd64__
unsigned long _lrotl(unsigned long __X, int __C)
{
    return fast_amd64_specific_lrotl(__X, __C);
}
#else
unsigned long _lrotl(unsigned long __X, int __C)
{
    return (__X << __C) | (__X >> ((sizeof(long) * 8) - __C));
}
#endif
```

- (Tipp: Anzeigen der vordefinierten Compileschalter:
`touch empty.c;<Plattform>-gcc -E -dM empty.c`)

„Eingebaute“ Compileschalter

Hochschule RheinMain

- Compiler definieren in der Regel eine (große) Anzahl von Compileschaltern vor, aus denen auf verschiedene Eigenschaften des verwendeten Compilers geschlossen werden kann.
- Damit können im Code Plattformabhängigkeiten automatisch angepasst werden.

Eingebauter Compileschalter: ist definiert, wenn der AMD64-Compiler verwendet wird

```
#ifdef __amd64__
unsigned long _lrotl(unsigned long __X, int __C)
{
    return fast_amd64_specific_lrotl(__X, __C);
}
#else
unsigned long _lrotl(unsigned long __X, int __C)
{
    return (__X << __C) | (__X >> ((sizeof(long) * 8) - __C));
}
#endif
```

- (Tipp: Anzeigen der vordefinierten Compileschalter:
`touch empty.c;<Plattform>-gcc -E -dM empty.c`)

„Eingebaute“ Compileschalter

Hochschule RheinMain

- Compiler definieren in der Regel eine (große) Anzahl von Compileschaltern vor, aus denen auf verschiedene Eigenschaften des verwendeten Compilers geschlossen werden kann.
- Damit können im Code Plattformabhängigkeiten automatisch angepasst werden.

```
#ifdef __amd64__
unsigned long _lrotl(unsigned long __X, int __C)
{
    return fast_amd64_specific_lrotl(__X, __C);
}
#else
unsigned long _lrotl(unsigned long __X, int __C)
{
    return (__X << __C) | (__X >> ((sizeof(long) * 8) - __C));
}
#endif
```

Eingebauter Compileschalter: ist definiert, wenn der AMD64-Compiler verwendet wird

Schnelle (Assembler-) Implementierung, nur für AMD64

- (Tipp: Anzeigen der vordefinierten Compileschalter:
  `touch empty.c;<Plattform>-gcc -E -dM empty.c`)

„Eingebaute“ Compileschalter

Hochschule RheinMain

- Compiler definieren in der Regel eine (große) Anzahl von Compileschaltern vor, aus denen auf verschiedene Eigenschaften des verwendeten Compilers geschlossen werden kann.
- Damit können im Code Plattformabhängigkeiten automatisch angepasst werden.

```
#ifdef __amd64__
unsigned long _lrotl(unsigned long __X, int __C)
{
    return fast_amd64_specific_lrotl(__X, __C);
}
#else
unsigned long _lrotl(unsigned long __X, int __C)
{
    return (__X << __C) | (__X >> ((sizeof(long) * 8) - __C));
}
#endif
```


- (Tipp: Anzeigen der vordefinierten Compileschalter:
`touch empty.c;<Plattform>-gcc -E -dM empty.c`)

Anwendung: assert

Hochschule RheinMain

- Das Makro `assert()`[2] ist in der Headerdatei `<assert.h>` (in etwa) so definiert:

```
#ifdef NDEBUG
#define assert(annahme)
#else
#define assert(annahme) if(!(annahme))\
{\
     printf("File %s, line %d: Assertion failed!\n", __FILE__, __LINE__);\
     exit(EXIT_FAILURE);\
}
#endif
```

- Falls der Compileschalter `NDEBUG` **nicht** gesetzt ist...
  - ... wird die `annahme` als logischer Ausdruck geprüft.
  - Trifft sie **nicht** zu, wird das Programm unter Angabe der Stelle im Quellcode abgebrochen.
- Ist `NDEBUG` gesetzt, so werden alle `assert` Macros entfernt
  → Kein Laufzeitaufwand.
- Bei Mikrocontroller-Plattformen ohne `printf()` oder sinnvolles `exit()` (z.B. Arduino) kann man auch ein eigenes `assert()` definieren (z.B. in einer Endlossschleife `__LINE__` X blinken).

[2] Engl. *to assert* = annehmen, voraussetzen.

Anwendung: assert

Hochschule RheinMain

- Das Makro assert()[2] ist in der Headerdatei <assert.h> (in etwa) so definiert:

```
#ifdef NDEBUG
#define assert(annahme)
#else
#define assert(annahme) if(!(annahme))\
{\
     printf("File %s, line %d: Assertion failed!\n", __FILE__, __LINE__);\
     exit(EXIT_FAILURE);\
}
#endif
```

Fortsetzung der Makrodefinition in der nächsten Zeile

- Falls der Compileschalter NDEBUG **nicht** gesetzt ist...
  - ... wird die annahme als logischer Ausdruck geprüft.
  - Trifft sie **nicht** zu, wird das Programm unter Angabe der Stelle im Quellcode abgebrochen.
- Ist NDEBUG gesetzt, so werden alle assert Macros entfernt
  → Kein Laufzeitaufwand.
- Bei Mikrocontroller-Plattformen ohne printf() oder sinnvolles exit() (z.B. Arduino) kann man auch ein eigenes assert() definieren (z.B. in einer Endlossschleife __LINE__ X blinken).

[2] Engl. *to assert* = annehmen, voraussetzen.

Anwendung: assert

Hochschule RheinMain

- Das Makro `assert()`[2] ist in der Headerdatei `<assert.h>` (in etwa) so definiert:

```
#ifdef NDEBUG
#define assert(annahme)
#else
#define assert(annahme) if(!(annahme))\
{\
     printf("File %s, line %d: Assertion failed!\n", __FILE__, __LINE__);\
     exit(EXIT_FAILURE);\
}
#endif
```

`__FILE__` wird durch den aktuellen Dateinamen ersetzt, `__LINE__` durch die aktuelle Zeilennummer

- Falls der Compileschalter `NDEBUG` **nicht** gesetzt ist...
  - ... wird die `annahme` als logischer Ausdruck geprüft.
  - Trifft sie **nicht** zu, wird das Programm unter Angabe der Stelle im Quellcode abgebrochen.
- Ist `NDEBUG` gesetzt, so werden alle `assert` Macros entfernt
  → Kein Laufzeitaufwand.
- Bei Mikrocontroller-Plattformen ohne `printf()` oder sinnvolles `exit()` (z.B. Arduino) kann man auch ein eigenes `assert()` definieren (z.B. in einer Endlossschleife `__LINE__` X blinken).

[2] Engl. *to assert* = annehmen, voraussetzen.

Anwendung: assert

Hochschule RheinMain

- Das Makro `assert()`[2] ist in der Headerdatei `<assert.h>` (in etwa) so definiert:

```
#ifdef NDEBUG
#define assert(annahme)
#else
#define assert(annahme) if(!(annahme))\
{\
     printf("File %s, line %d: Assertion failed!\n", __FILE__, __LINE__);\
     exit(EXIT_FAILURE);\
}
#endif
```

Compileschalter zur Aktivierung

- Falls der Compileschalter `NDEBUG` **nicht** gesetzt ist...
  - ... wird die `annahme` als logischer Ausdruck geprüft.
  - Trifft sie **nicht** zu, wird das Programm unter Angabe der Stelle im Quellcode abgebrochen.
- Ist `NDEBUG` gesetzt, so werden alle `assert` Macros entfernt
  → Kein Laufzeitaufwand.
- Bei Mikrocontroller-Plattformen ohne `printf()` oder sinnvolles `exit()` (z.B. Arduino) kann man auch ein eigenes `assert()` definieren (z.B. in einer Endlossschleife `__LINE__` X blinken).

[2] Engl. *to assert* = annehmen, voraussetzen.

Anwendung: assert

Hochschule RheinMain

- Das Makro `assert()`[2] ist in der Headerdatei `<assert.h>` (in etwa) so definiert:

```
#ifdef NDEBUG
#define assert(annahme)
#else
#define assert(annahme) if(!(annahme))\
{\
     printf("File %s, line %d: Assertion failed!\n", __FILE__, __LINE__);\
     exit(EXIT_FAILURE);\
}
#endif
```

Compileschalter zur Aktivierung

- Falls der Compileschalter `NDEBUG` **nicht** gesetzt ist...
  - ... wird die `annahme` als logischer Ausdruck geprüft.
  - Trifft sie **nicht** zu, wird das Programm unter Angabe der Stelle im Quellcode abgebrochen.
- Ist `NDEBUG` gesetzt, so werden alle `assert` Macros entfernt
  → Kein Laufzeitaufwand.
- Bei Mikrocontroller-Plattformen ohne `printf()` oder sinnvolles `exit()` (z.B. Arduino) kann man auch ein eigenes `assert()` definieren (z.B. in einer Endlossschleife `__LINE__` X blinken).

[2] Engl. *to assert* = annehmen, voraussetzen.

Anwendung: assert

Hochschule RheinMain

- Das Makro assert()[2] ist in der Headerdatei <assert.h> (in etwa) so definiert:

```
#ifdef NDEBUG
#define assert(annahme)
#else
#define assert(annahme) if(!(annahme))\
{\
     printf("File %s, line %d: Assertion failed!\n", __FILE__, __LINE__);\
     exit(EXIT_FAILURE);\
}
#endif
```

Compileschalter zur Aktivierung

- Falls der Compileschalter NDEBUG **nicht** gesetzt ist...
  - ... wird die annahme als logischer Ausdruck geprüft.
  - Trifft sie **nicht** zu, wird das Programm unter Angabe der Stelle im Quellcode abgebrochen.
- Ist NDEBUG gesetzt, so werden alle assert Macros entfernt
  → Kein Laufzeitaufwand.
- Bei Mikrocontroller-Plattformen ohne printf() oder sinnvolles exit() (z.B. Arduino) kann man auch ein eigenes assert() definieren (z.B. in einer Endlossschleife __LINE__ X blinken).

[2] Engl. *to assert* = annehmen, voraussetzen.

assert() Anwendungsbeispiele

Hochschule RheinMain

```
#include <assert.h>
....
int stringlaenge(char *string)
{
    int i;
    assert(string != NULL);
    for(i = 0; *string != 0; i++)
        ;
    return i;
}
```

```
#include <assert.h>
....
#define ANZAHL_WERTE 4711
....
int Array[ANZAHL_WERTE];
....
void wert_speichern(int wert, int index)
{
    assert(index >= 0 && index < ANZAHL_WERTE);
    Array[index] = wert;
}
```

assert() Anwendungsbeispiele

Hochschule RheinMain

```
#include <assert.h>
....
int stringlaenge(char *string)
{
    int i;
    assert(string != NULL);
    for(i = 0; *string != 0; i++)
        ;
    return i;
}
```

Vor Dereferenzierung prüfen, ob string nicht NULL (d.h. ungültig) ist

```
#include <assert.h>
....
#define ANZAHL_WERTE 4711
....
int Array[ANZAHL_WERTE];
....
void wert_speichern(int wert, int index)
{
    assert(index >= 0 && index < ANZAHL_WERTE);
    Array[index] = wert;
}
```

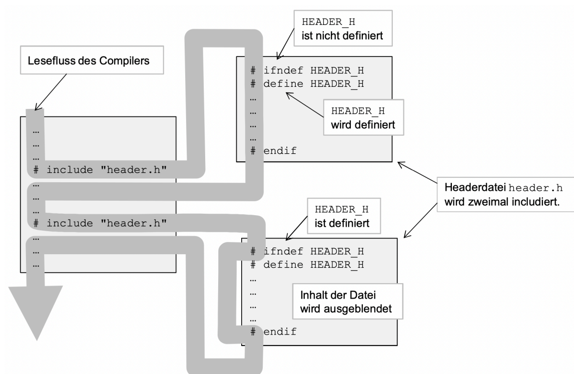
Vor dem Schreiben sicherstellen, dass index im zulässigen Bereich liegt

Schutz vor rekursivem Include

Hochschule RheinMain

,

- Mit Compileschaltern kann verhindert werden, dass eine Headerdatei mehrfach includiert wird:



- Wird die Headerdatei erstmalig includiert ist der Compileschalter noch nicht gesetzt. Die Headerdatei ist also für den Compiler sichtbar. In der Headerdatei wird dann der Compileschalter gesetzt, sodass die Datei, bei weiteren Includes ausgeblendet wird.

Ein kleines Projekt mit drei Dateien

Hochschule RheinMain

maximum.h

```
# ifndef MAXIMUM_H
# define MAXIMUM_H

extern int absolutes_maximum;
extern int maximum( int a, int b);

# endif
```

Hier werden die globale Variable `absolutes maximum` und die Funktion `maximum` **deklariert**.

main.c

```
# include <stdio.h>
# include <stdlib.h>
# include "maximum.h"

void main()
    {
    int a, b, c;

    a = maximum( 1, -5);
    b = maximum( -10, 17);
    c = absolutes_maximum;
    printf( "%d\n", c);
    }
```

Hier werden die globale Variable `absolutes maximum` und die Funktion `maximum` **verwendet**.

maximum.c

```
# include <limits.h>
# include "maximum.h"

int absolutes_maximum = INT_MIN;

int maximum( int a, int b)
    {
    int max;

    if( a > b)
        max = a;
    else
        max = b;
    if( max > absolutes_maximum)
        absolutes_maximum = max;
    }
```

Hier werden die globale Variable `absolutes maximum` und die Funktion `maximum` **definiert**.

`INT_MIN` ist eine symbolische Konstante, die in `limits.h` als der kleinstmöglche `int`-Wert festgelegt ist.

- Headerdateien enthalten Deklarationen, die in unterschiedlichen Quelldateien konsistent verwendet werden sollen. Headerdateien enthalten keine Definitionen und keinen Code, sie ermöglichen nur die Aufteilung von Definitionen und Code auf mehrere Dateien.