Das erste Programm (Folie 16)

- Kommentare werden vom Compiler vollständig ignoriert.
- Elimination durch Präprozessor

```
//  Kommentar bis zum Zeilenende
/*  Diese Art von Kommentar kann
    beliebig viele Zeilen lang sein und
    und endet mit der nächsten Zeichen-
    folge der Form */
```

- Passende und geeignete Kommentare sind für die Lesbarkeit von Programmen von zentraler Bedeutung.

Das erste Programm (Folie 16)

#include<iostream>

- Einbindung der Ein-/Ausgabefunktionen (von Objekten)
  - cin : Standardeingabekanal
  - cout : Standardausgabekanal
  - cerr : Fehlerkanal
  - endl : Zeilentrenner
- Operatoren auf iostream-Objekten
  - << : „Schiebe die Daten zur Ausgabe“ (cout, cerr)
  - >> : „Lies Daten in eine Variable ein“ (cin)
- Anweisung an den Präprozessor
- Textueller Import von externen „Header-Dateien“
  - Inhalt der Header-Datei ersetzt diese Zeile
- C++ Module/Funktionen/Klassen müssen vor Benutzung importiert/includiert werden.
- Zeilen, die mit # beginnen, enthalten Präprozessor/Compiler-Anweisungen.

Das erste Programm (Folie 16)

using namespace std;

- Der Namensraum „std“ wird benutzt.
- Alle Symbole der Standard C++ Bibliothek sind im Namespace „std“ definiert.
- Schreiben Sie diese Zeile in jedes Programm an diese Stelle.
- Genauere Erklärung folgt später.

Breymann_Folien

Das erste Programm (Folie 16)

main

- ist das Schlüsselwort für das Hauptprogramm (später werden wir auch Unterprogramme und Funktionen kennenlernen).
- Alle Schlüsselwörter werden in C++ klein geschrieben.

main ( )

- In der runden Klammer können, falls erforderlich, Parameter an das Hauptprogramm übergeben werden.

int main ( )

- Das Hauptprogramm kann nach Beendigung eine ganze Zahl (Datentyp int) an das Betriebssystem zurückgeben.
- Bei ordnungsgemäßem Programmablauf wird die Zahl 0 zurückgegeben : int main ( ) { ..... return 0;}

int main ( ) { .....}

- Der zu dem Programm gehörende Programmcode wird durch die geschweiften Klammern { und } eingeschlossen.
- Ein mit { und } begrenzter Bereich heißt Block.

Breymann_Folien

Das erste Programm (Folie 16)

```
int  summe, a, b;
```

- Deklaration und Definition von 3 Variablen: summe, a, b vom Datentyp „int“ (ganze Zahl der Größe 32 Bit).
- Der Compiler reserviert für jede dieser Variablen einen wohldefinierten Speicherplatz der Größe 32 Bit.
- Mit Hilfe (des Namens) der Variablen können wir auf diesen Speicherplatz zugreifen und ihn manipulieren.

```
a = 10;
b = 5;
summe = a + b;
```

- Der Programmierer wählt die Namen der Variablen unter Verwendung einer vorgegebenen Namenskonvention.
- Wir könnten die Definition auch umständlicher schreiben als :

```
int summe;
int a;
int b;
```

Breymann_Folien

Das erste Programm (Folie 16)

```
int  summe, a, b;

datentyp   variablenname;
datentyp   vname1, vname2, ....., vnameX;
```

Namenskonvention (siehe Breymann Seite 37):

- Ein Name ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und Unterstrichen „_“.
- Ein Name beginnt immer mit einem Buchstaben oder einem Unterstrich (letzteres sollte jedoch vermieden werden).
- Selbsterfundene Namen sollten nicht mit den vordefinierten Schlüsselwörtern (z.B „main“, „int“ usw.) übereinstimmen.
- Meist wird die Länge der Namen durch die Compiler begrenzt (31 oder 255). Prinzipiell kann ein Name beliebig lang sein.

Breymann_Folien

Das erste Programm (Folie 16)

- Regeln für die Verwendung von Variablen:
  - Alle Variablen müssen vor Benutzung deklariert werden.
  - Jede benutzte Variable muss genau einmal im Programm definiert werden.
  - Definitionen und Deklarationen müssen nicht am Anfang eines Code-Blocks stehen (wie in C).
- Empfehlung:
  - Passende Namen erleichtern das Verständnis des Programms und machen es „lesbarer“.

Breymann_Folien

Das erste Programm (Folie 16)

- Einfache Datentypen werden in der nächsten Vorlesung (Stunde) ausführlich diskutiert.
  - Datentypen für ganze Zahlen:
    - short 16 Bit unsigned short
    - int 32 Bit (16 Bit?) unsigned int
    - long 64 Bit (32 Bit?) unsigned long
  - Datentypen für Gleitkommazahlen:
    - float 32 Bits +/- 3.4 * $10^{+/-38}$
    - double 64 Bits +/- 1.7 * $10^{+/-308}$
    - long double 80 Bits +/- 3.4 * $10^{+/-4932}$
  - Für Zeichen, Text und Strings:
    - char 8 Bit unsigned char
    - string „dynamisch“ #include<string> // „string“-Header
  - Logischer Datentyp:
    - bool kann nur zwei Werte „true“ oder „false“ annehmen.

Breymann_Folien

Das erste Programm (Folie 16)

- Man kann auch Konstante deklarieren und initialisieren:

```
const  float  pi  =  3.14159254;     // Die Zahl  pi
```

- Der Compiler reserviert Speicherplatz und initialisiert den Speicherplatz mit dem angegebenen Wert, der in dem entsprechenden Programm nicht verändert werden kann.
- Auch Variablen können direkt in (nach) der Definition initialisiert werden:

```
int      zaehler = 0;
float    flaeche = 10.0 * 10.0;
string   programm_name("mein erstes Programm");
```

Breymann_Folien

Das erste Programm (Folie 16)

```
cout  <<  "a und b eingeben :";
```

- Ausgabe: cout (Abkürzung für character out) ist die Standardausgabe (in der Regel auf dem Bildschirm).
- Der „<<„ Doppelpfeil deutet an, dass alles, was rechts davon steht, zur Ausgabe cout gesendet wird.
- Beliebige Texte können wir oben angegeben auf die Standardausgabe ausgeben (zu Fragen der Formatierung siehe Breymann Folie 34).

```
cout << "Summe="  <<  summe;
```

- Sollen mehrere Dinge ausgegeben werden, so sind sie durch „<<„ zu trennen.
- Falls wir für a und b die Werte 10 und 5 eingeben, so sieht die Ausgabe der obigen Zeile auf dem Bildschirm wie folgt aus:

```
Summe=15
```

Breymann_Folien

Das erste Programm (Folie 16)

```
cin  >>  a   >>  b;
```

- Eingabe: cin (Abkürzung für character in) ist die Standardeingabe (in der Regel mit der Tastatur).
- Der „>>„ Doppelpfeil zeigt hier in Richtung des Objekts, das von der Tastatur einen neuen Wert aufnehmen soll.
- Die Information fließt von der Eingabe cin zum Objekt a bzw. b.

```
int  a;
float  x;
string  text;
cin >> a  >> x  >> text;
```

- cin überliest Zwischenräume (Whitespace) z.B.
  - Leerzeichen
  - Tabs
  - Newline
- Liest dann Zeichen bis wieder Whitespace auftaucht.

Breymann_Folien

Das erste Programm (Folie 16)

```
summe = a + b;
```

- Die Summe der Werte der beiden Variablen a und b wird berechnet und „summe“ zugewiesen, d.h., unter dem für „summe“ reservierten Speicherplatz gespeichert.

```
summe = a + b
```

- ist ein sogenannter Ausdruck.
- Ein Ausdruck besteht aus Operanden (summe, a, b), die miteinander durch Operatoren (+, =) verknüft sind.
- Die Definition von Ausdrücken wird später ausführlich behandelt.

Breymann_Folien

Das erste Programm (Folie 16)

return 0;

- Dies ist die letzte Zeile im Hauptprogramm.
- Sie ist im Skript nicht angegeben.
- Das Hauptprogramm liefert nach Beendigung an das Betriebssystem eine ganze Zahl zurück, welche Auskunft über den regulären Ablauf (0) gibt.
- Falls Probleme oder Fehler auftreten, kann man das Hauptprogramm auch vorher abbrechen:

return -1;

Breymann_Folien

Das erste Programm (Folie 16)

Falls unser Beispielprogramm in der Datei „summe.cpp“ gespeichert ist, dann können wir es wie folgt übersetzen:

```
g++  -c   summe.cpp
```

Hierbei wird (falls der Compiler keine Fehler meldet) die Datei „summe.o“ erzeugt.

```
g++  -o  summe  summe.o
```

Der Linker erzeugt die ausführbare Datei mit dem Namen „summe“ (unter Windows würde man den Namen „summe.exe“ wählen).

Die beiden oberen Zeilen können zu einer Zeile zusammengefasst werden:

```
g++  -o  summe  summe.cpp
```

Das Programm kann durch Eintippen von „summe“ gestartet werden.

Falls Header nicht gefunden werden, dann kann man die Namen der entsprechenden Verzeichnisse wie folgt angeben:

```
g++  -I/Pfad_zum_Verzeichnis/include  -o summe summe.cpp
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Deklarationsanweisung: Führt einen Namen in ein Programm ein:

```
int summe;
const float pi = 3.1415;
string  text;
```

- Nach Deklarationen sind die Namen im Programm bekannt und können verwendet werden.

- Eine einfache Deklaration wird stets mit einem Semikolon abgeschlossen.

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Eine Ausdrucksanweisung ist ein Ausdruck gefolgt von einem Semikolon:

```
a = b;
summe = a + b;
```

- Ein Ausdruck repräsentiert nach der Auswertung einen Wert.

- Auch die Anweisungen

```
cin >>  a  >>  b;
cout <<  summe;
```

sind Ausdrucksanweisungen.

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Eine Verbundanweisung, auch Block genannt, ist ein Paar von geschweiften Klammer { und }, die eine Folge von Anweisungen enthalten:

```
{   }                          //  leerer Block

{                              //  Block mit einer Anweisung
  Anweisung1
}

{                              //  Block mit mehreren Anweisungen
  Anweisung1
  Anweisung2
  .....
}
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Eine Kontroll- oder Verzweigungsanweisung erlaubt es, die Ausführung von Anweisungen von Bedingungen abhängig zu machen:

```
Falls          ich heute Abend noch Zeit habe
dann           setze ich noch die erste Übung auf
andernfalls    werde ich die Übung morgen anfertigen
```

- Die if-Anweisung ermöglicht diese Art von Fallunterscheidungen:

```
if  (Bedingung)  Anweisung1

if (Bedingung)  Anweisung1
else            Anweisung2
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Auch Blöcke von Anweisungen können in Abhängigkeit von der Bedingung ausgeführt werden:

```
if  (Bedingung)
    {
       Anweisung1
       Anweisung2
       ....
     }
else
   {
      Anweisung_else_1
      Anweisung_else_2
      .....
   }
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Das folgende Programm berechnet den Betrag der Differenz von a und b: ||a-b||

```
#include<iostream>
using namespace std;

int main()  {
   int  betrag, a, b;
   cout << "a und b eingeben:";      // Lies die Zahlen a und b ein
   cin >>  a  >>  b;

   if  (a >= b)   betrag  =  a  - b;      //  Berechne den Betrag
   else           betrag  =  b – a;

   cout <<  "Betrag(a-b) = "  <<  betrag  << endl;
   return 0;
}
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Umwandlung römischer Ziffern

```
#include<iostream>
using namespace std;

int main()  {
  int  a = 0;                          // Variable für das Resultat
  char c;                              // Variable zum Einlesen der Ziffer
  cout <<  "Ziffer :" ;                //  Eingabeaufforderung
  cin  >> c;                           //  Einlesen der Ziffer

  if  (c == 'I')        a = 1;         // Berechnung der arabischen Zahlen
  else if (c == 'V')  a = 5;
  else if (c == 'X')  a = 10;
  else if (c == 'L')   a = 50;
  else if (c == 'C')  a = 100;
  else if (c == 'D')  a = 500;
  else if (c == 'M')  a = 1000;


  if  (a == 0)  cout << "keine römische Ziffer!\n" << endl;     // Ausgabe des Resultats
  else          cout <<  a  << endl;

  return 0;
}
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Die wichtigsten Vergleichsoperatoren:

==	gleich	if (a == b)	if ((a-b) == 0)
!=	ungleich	if (a != b)	
>	größer	if (a > b)	
<	kleiner	if (a < b)	
>=	größer gleich	if (a >= b)	
<=	kleiner gleich	if (a <= b)	

Wir werden später noch mehr Vergleichsoperatoren kennen lernen. Jeder Datentyp besitzt eine spezielle Menge von solchen Operatoren. Siehe Breymann-Folien Seite 22--38.

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Mehrere Bedingungen können durch logische UND- oder ODER-Operatoren kombiniert werden:

&&	Logisches UND
||	Logisches ODER
!	Logische Negation

```
if  ( a > 0 ||  a < 0)   cout <<  " a ist ungleich 0" << endl;
else                     cout <<  " a ist gleich 0"  << endl;

char c = 'B';
bool  grossbuchstabe;

if ( (c >= 'A') && (c <= 'Z'))   grossbuchstabe = true;
else                             grossbuchstabe = false;
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- If-Anweisungen können natürlich geschachtelt werden:

```
int a, b, c, maximum;
cin >> a >> b >> c;

// Bestimme das Maximum von a, b und c
if ( a >= b)
   {
     if ( c >= a)  maximum = c;
     else          maximum = a;
   }
else
   {
     if (c >= b)  maximum = c;
     else         maximum = b;
   }
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- switch-Anweisungen erlauben die übersichtliche Auswahl (Fallunterscheidung) unter vielen Anweisungen:

```
switch(Ausdruck) {
   case  const1 : Anweisungen  break;
   case  const2 : Anweisungen  break;
   // ...
   default          : ErsatzAnweisungen
}
```

- Der Ausdruck wird ausgewertet und muss ein Ergebnis vom Typ „int“ haben oder leicht in „int“ konvertierbar zu sein, wie zum Beispiel „char“.
- Das Ergebnis des Ausdrucks wird mit den case-Konstanten verglichen, die zum Einsprung an die richtige Stelle dienen.

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Umwandlung römischer Ziffern

```
#include<iostream>
using namespace std;

int main()  {
  int  a = 0;                          // Variable für das Resultat
  char c;                              // Variable zum Einlesen der Ziffer
  cout <<  “Ziffer :“ ;                //  Eingabeaufforderung
  cin  >> c;                           //  Einlesen der Ziffer

  if  (c == ‘I’)         a = 1;        // Berechnung der arabischen Zahlen
  else if (c == ‘V’)  a = 5;
  else if (c == ‘X’)  a = 10;
  else if (c == ‘L‘)   a = 50;
  else if (c == ‘C‘)  a = 100;
  else if (c == ‘D’)  a = 500;
  else if (c == ‘M‘)  a = 1000;


  if  (a == 0)  cout << “keine römische Ziffer!\n“ << endl;     // Ausgabe des Resultats
  else          cout <<  a  << endl;

  return 0;
}
```

Breymann_Folien

Kontrollstrukturen: Anweisungen

- Umwandlung römischer Ziffern mittels switch-case-Variationen:

```
#include<iostream>
using namespace std;

int main()  {
  int  a;                                          // Variable für das Resultat
  char c;                                          // Variable zum Einlesen der Ziffer
  cout <<  "Ziffer :" ;                            //  Eingabeaufforderung
  cin  >> c;                                       //  Einlesen der Ziffer

  switch (c)   {                                   // Berechnung der arabischen Zahlen
               case  'I'   :   a = 1;       break;
               case  'X'  :   a = 10;      break;
               case  'L'  :   a = 50;      break;
               case  'C'  :   a = 100;    break;
               case  'D'  :   a = 500;    break;
               case  'M'  :   a = 1000;  break;
               default    :    a = 0;
  }


  if  (a == 0)  cout << "keine römische Ziffer!\n" << endl;    // Ausgabe des Resultats
  else          cout <<  a  << endl;

  return 0;
}
```

Breymann_Folien

Schleifenanweisungen: while-Schleife

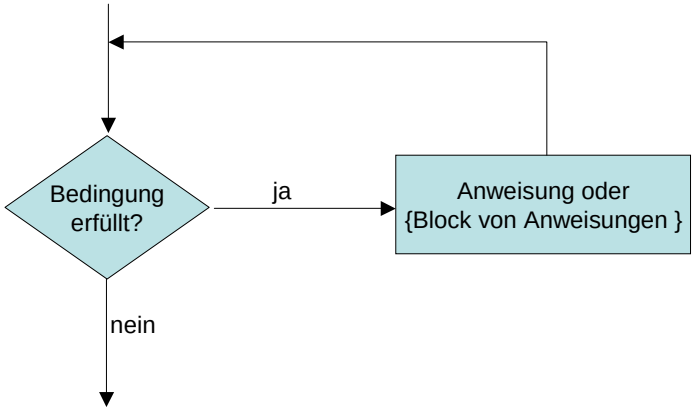
- In vielen Anwendungen muss die gleiche Aufgabe oft wiederholt werden.
- Hierfür gibt es in C++ so genannte Schleifen.
- Nach Abarbeitung des Schleifenblocks beginnt man mit der Abarbeitung wieder am Blockanfang.
- Dies geschieht solange, wie die Schleifenbedingung erfüllt ist.
- Syntax von while-Schleifen:

```
while (Bedingung)   Anweisung
while (Bedingung)  {  Block_von_Anweisungen }
```

Breymann_Folien

Schleifenanweisungen: while-Schleife

Flussdiagramm für eine while-Anweisung



Auch while-Anweisungen können geschachtelt werden.

Breymann_Folien

Schleifenanweisungen: while-Schleife

Beispiel für while-Schleife: Fakultät

```
int  n = 2, grenze;
long fak = 1;
cin >> grenze;

while ( n <= grenze)
    {
        fak = fak * n;      // kurz:   fak *= n;
        n++;                // entspricht  n = n +1;
    }
```

Breymann_Folien

Schleifenanweisungen: while-Schleife

Neue Operatoren:

Binäre

```
+=    Beispiel:  a += b;   entspricht  a = a + b;
-=    Beispiel:  a -=  b;  entspricht  a = a – b;
*=    Beispiel   a *=  b;  entspricht  a = a * b;
/=    Beispiel   a  /=  b; entspricht  a = a / b;
%=    Beispiel   a %= b;   entspricht  a = a%b;   modulo nur für
                                                  ganzzahlige Typen
```

Unäre

```
++    Beispiel   n++;      entspricht  n = n+1;
                 ++n;      entspricht  n = n+1;
--    Beispiel   n--;      entspricht  n = n-1;
                 --n;      entspricht  n = n-1;
```

Was ist der Unterschied zwischen vor- und nachgestelltem Operator (++, --)?
Man kann diese unären Operatoren wie folgt in Ausdrücken verwenden:

```
while ( n <= grenze)  fak *= n++;    // gleich:  fak *= n; n = n + 1;
while ( n <= grenze)  fak *= ++n;    // gleich:  n = n + 1; fak *= n;
```

Breymann_Folien

Schleifenanweisungen: while-Schleife

Berechnung des größten gemeinsamen Teilers GGT(x,y):

- GGT(x,x) = x
- GGT(x,y) = GGT(x,y-x), falls x<y;

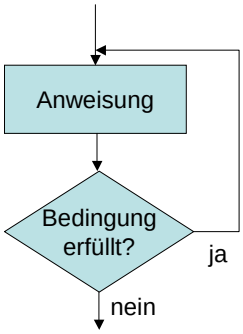
```
#include<iostream>
using namespace std;

int main( )  {
unsigned int x, y;

cout  << "2 Zahlen größer 0 eingeben:";
cin >> x >> y;
cout  << " Der GGT von "  <<  x  <<  " und "  << y << " ist: ";

while ( x != y )
      if ( x > y)  x -= y;      //  if ... else zählt als eine Anweisung
      else         y -= x;      //  deshalb ohne {  }

cout  <<  x   <<   endl;
}
```

Breymann_Folien

Schleifenanweisungen: while-Schleife

- C++ stellt auch so genannte do-while-Schleifen zur Verfügung:
- Syntax:

```
do
    Anweisung  oder
    { Block_von_Anweisungen }
while (Bedingung)
```



Breymann_Folien

Schleifenanweisungen: for-Schleife

- for-Schleifen erlauben eine sehr kompakte und übersichtliche Eingabe von Initialisierungen, Bedingungen und Veränderungen.
- Syntax der for-Schleife:

```
for (Initialisierung ; Bedingung ; Veränderung)  Anweisung oder {Block}
```

- Beispiel:

```
int  n, grenze;
long fak;
cin >> grenze;
for ( n = 2, fak = 1 ;  n <= grenze ; n++)   fak *= n;
```

- Bedeutung:
  - Durchführung der Initialisierung: Setze Startwerte. Diese Anweisung(en) werden nur einmal am Start durchgeführt.
  - Überprüfe die Bedingung. Falls Sie erfüllt ist, dann führe die Anweisung (oder Anweisungen im Block) durch. Anschließend die Veränderung(en).
  - Dieser Vorgang wird wiederholt bis die Bedingung nicht mehr erfüllt ist.

Breymann_Folien

Schleifenanweisungen: for-Schleife

- Der Zähler kann auch in der Initialisierung der for-Schleife definiert werden:

```
fak = 1;
for (int  n = 2 ; n <= grenze ; ++n)  fak *= n;
```

- Die Variable n ist dann nur innerhalb der for-Schleife „bekannt“ (definiert).
- Mittels der „break“-Anweisung kann man aus einer for-Schleife (while-Schleife, switch-Anweisung) aussteigen (die Ausführung beenden).

```
for ( ... ;  ... ;  ...)
   {
     ...
     if ( ...)  break;
     ...
   }
```

- Man beachte, dass die „break“-Anweisung bei geschachtelten Schleifen nur die Ausführung der innersten Schleife beendet.

Breymann_Folien

Schleifenanweisungen: for-Schleife

- Mittels der „continue“-Anweisung kann man die Ausführung von bestimmten Teilen der Schleife (alle Anweisungen, die der „continue“-Anweisung folgen) überspringen.

```
for ( ... ;  ... ;  ...)
   {
     ...
     if ( ...)  continue;
     ...
     ...
   }
```

wird gegebenfalls nicht ausgeführt !!!

- In einer for-Schleife geht man sofort zur Ausführung der Veränderung(en) und dann wieder zur Überprüfung der Bedingung über.

Breymann_Folien

Schleifenanweisungen: for-Schleife

- Den Kommaoperator hatten wir bereits bei der Initialisierung einer for-Schleife kennengelernt.

```
for (int  n = 2 , fak = 1 ; n <= grenze ; fak *= n, ++n)  ;
```

- Hier haben wir die Anweisung im Schleifenkörper in die Veränderung integriert.

- Dieser Programmierstil führt nicht zu leicht verständlichen Programmen und ist deshalb nicht empfehlenswert.

Breymann_Folien

Unterprogramme/Funktionen

- Große Programme müssen in übersichtliche Teile zerlegt werden:
  - Delegieren von Teilaufgaben in Form von Funktionen
  - Strukturieren und Zusammenfassen von zusammengehörigen Aufgaben in Modulen (Wiederverwendbarkeit)

```
#include<iostream>
using namespace std;

int main( )  {
        unsigned int x, y;

        cout  << "2 Zahlen größer 0 eingeben:";
        cin >> x >> y;
        cout  << " Der GGT von "  <<  x  <<  " und "  << y << " ist: ";

        while ( x != y )
                if ( x > y)  x -= y;       //  if ... else zählt als eine Anweisung
                else         y -= x;       //  deshalb ohne {  }

        cout  <<  x   <<   endl;
}
```

Breymann_Folien

Unterprogramme/Funktionen

```
#include<iostream>
using namespace std;

unsigned int ggt( unsigned int, unsigned int);
int main( ) {
        unsigned int x, y, resultat;

        cout << "2 Zahlen größer 0 eingeben:";
        cin >> x >> y;
        cout << " Der GGT von " << x << " und " << y << " ist: ";

        resultat = ggt(x,y);
        cout << "GGT(" << x << "," << y << ")=" << resultat << endl;
}

unsigned int ggt( unsigned int zahl1, unsigned int zahl2) {
        while ( zahl1 != zahl2 )
                if ( zahl1 > zahl2) zahl1 -= zahl2;
                else                zahl2 -= zahl1;
        return zahl1;
}
```

Breymann_Folien

Unterprogramme/Funktionen

- Die Funktionsdeklaration (Funktionsprototyp) gibt die Existenz einer entsprechenden Funktion bekannt.
- Syntax der Funktionsdeklaration/Funktionsprototyp:

Rückgabetyp Funktionsname (Parameterliste);

Unser Beispiel : Rückgabetyp = unsigned int
Funktionsname = ggt
Parameterliste = (unsigned int, unsigned int)

weitere Beispiele (für Parameterlisten):

leere Liste:	int	func(); = int func(void);
ein Param.	float	square_root(float);
zwei Param.	int	addition(int, int);
drei Param.	void	xyz(float, int, char);
usw.		

Breymann_Folien

Unterprogramme/Funktionen

- Eine Funktionsdefinition (Implementation) veranlasst den Compiler entsprechenden Code zu erzeugen.
- Syntax der Funktionsdefinition (Implementation):

Rückgabetyp Funktionsname (Formalparameterliste) Block

Unser Beispiel :

```
unsigned int  ggt( unsigned int zahl1, unsigned int zahl2) {
        while ( zahl1 != zahl2 )
                if ( zahl1 > zahl2)  zahl1 -= zahl2;
                else                 zahl2 -= zahl1;
        return zahl1;
}
```

- Hier müssen zwingend Namen für die Eingabeparameter angegeben werden, mit denen dann auch im Block gearbeitet werden kann.
- Die Namen der Eingabeparameter sind frei wählbar und völlig unabhängig vom Aufruf.
- Die Rückgabe des Resultats erfolgt mit der „return“-Anweisung.

Breymann_Folien

Unterprogramme/Funktionen

- Syntax des Funktionsaufrufs:

Funktionsname (Aktualparameterliste);

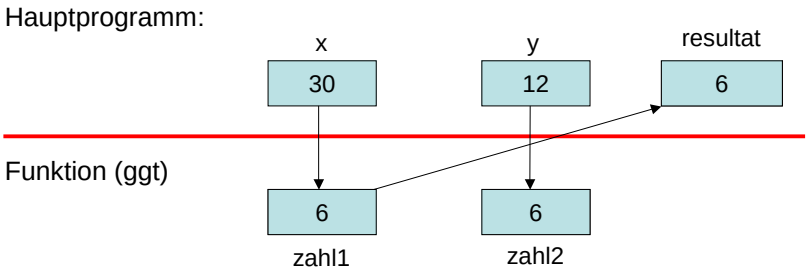
Unser Beispiel : resultat = ggt(x, y);

- Die Aktualparameterliste enthält die Namen der Variablen (Objekte) oder Audrücke, die an die Funktion übergeben werden sollen und für die die Funktion ausgeführt werden soll.
- Die Datentypen, die Zahl und Reihenfolge der Parameter müssen natürlich mit der Funktionsdeklaration/Funktions-definition übereinstimmen.

Breymann_Folien

Unterprogramme/Funktionen

- In unserem Beispiel erfolgte die Parameterübergabe per Wert:



- Später werden wir andere (effizientere) Möglichkeiten der Parameterübergabe kennen lernen.

Unterprogramme/Funktionen

- Funktionen können sich selbst rekursiv aufrufen oder andere ihnen bekannte Funktionen aufrufen.
  - Beispiel: Rekursive Funktion zur Berechnung von n! (n fakultaet)

```
long  fakultaet( long n) {
    if ( n == 1) return 1;
    else return( n * fakultaet(n-1));
}
```

- In obigem Beispiel ist der Aktualparameter vom Datentyp „long“ ein Ausdruck „n-1“ von diesem Datentyp.
- Auch „return“ können wir als Funktion interpretieren, die Ausdrücke vom richtigen Datentyp akzeptiert.
- Die obige Implementation ist ineffizient, da Funktionsaufrufe „teuer“ sind.

Modulare Gestaltung

- Es empfiehlt sich große Programme in einzelne, getrennt übersetzbare Dateien aufzuteilen (Wiederverwendbarkeit?).
- Standard-Header haben die Form #include<....>
- Eigene (und fremde, nicht-standard) Header-Dateien können hinzugefügt werden (siehe folgendes Beispiel).

```
#include “meine_mathe_funktionen.h“
```

- Die entsprechende Header-Datei mit dem Namen „meine_mathe_funktionen.h“ könnte wie folgt aussehen:

```
// meine_mathe_funktionen.h
unsigned int  ggt( unsigned int, unsigned int);
```

- Im Header findet man hier nur die Funktionsdeklaration.

Breymann_Folien

Modulare Gestaltung

- Die Einbindung in das Hauptprogramm erfolgt wie bei einem Standard-Header:

```
#include<iostream>
#include “meine_mathe_funktionen.h“
using namespace std;

int main( )  {
    unsigned int x, y, resultat;

    cout  << “2 Zahlen größer 0 eingeben:“;
    cin >> x >> y;
    cout  << “ Der GGT von “  <<  x  <<  “ und “  << y << “ ist: “;

    resultat = ggt(x,y);
    cout  <<  “GGT(“ << x << “,“ <<  y << “)=“ <<  resultat   <<   endl;
}
```

Breymann_Folien

Modulare Gestaltung

- Die Implementation der entsprechenden Funktion(en) findet man in der Datei „meine_mathe_funktionen.cpp“:

```
// meine_mathe_funktionen.cpp
#include “meine_mathe_funktionen.h“

unsigned int  ggt( unsigned int zahl1, unsigned int zahl2) {
        while ( zahl1 != zahl2 )
                if ( zahl1 > zahl2)  zahl1 -= zahl2;
                else                 zahl2 -= zahl1;
        return zahl1;
}
```

- Bevor das Hauptprogramm (Datei „berechne_ggt.cpp“) gelinkt werden kann, muss „meine_mathe_funktionen.cpp“ übersetzt werden:

```
g++  -c  meine_mathe_funktionen.cpp
g++  -c  berechne_ggt.cpp
g++  -o  berechne_ggt   berechne_ggt.o  meine_mathe_funktionen.o
```

- Das ausführbare Programm hat den Namen „berechne_ggt“.

Breymann_Folien