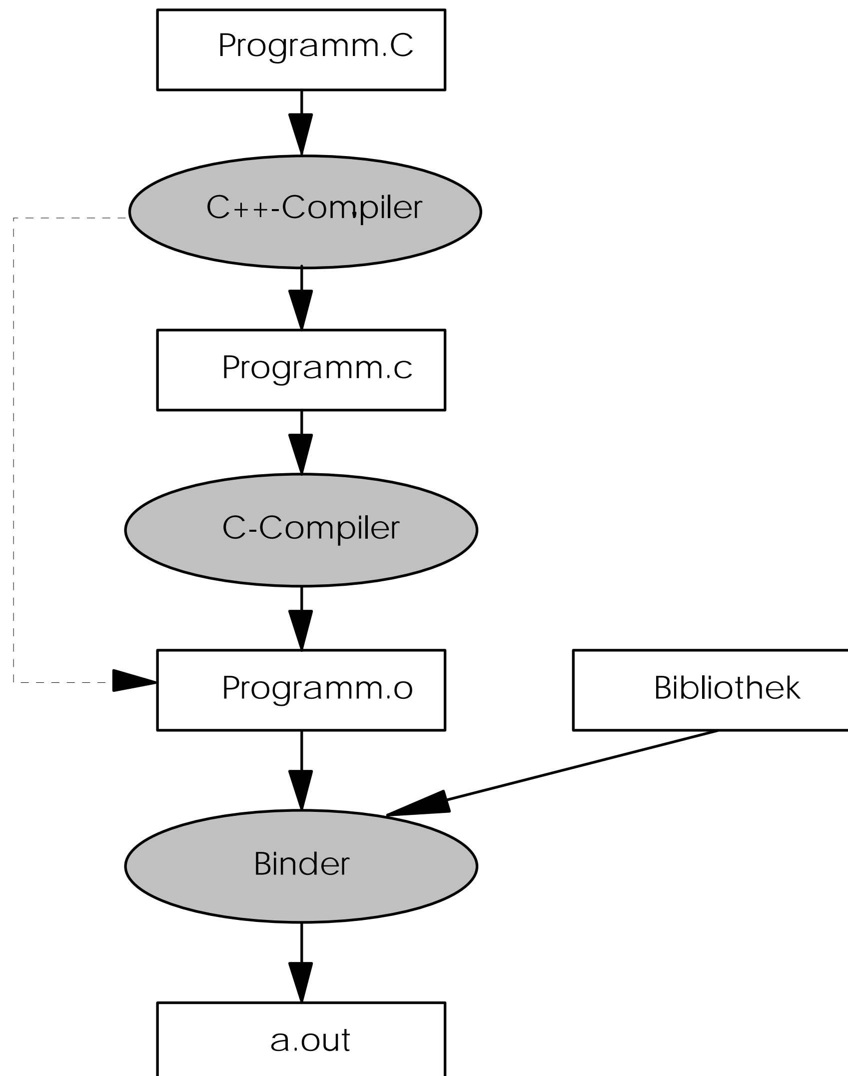
2 C++ als Erweiterung von C

2.1 Compiler

Zunächst wurden C++-Compiler als **Precompiler** (Translatoren) entwickelt, welche ein C++-Programm in ein reines C-Programm verwandeln. Diese wurden dann mit dem Standard-C-Compiler und mit den C-Bibliotheken zu einem ausführbaren Programm gebunden, so dass eine größtmögliche Kompatibilität erreicht wurde.



Inzwischen sind auch C++-Compiler entwickelt wurden, die ohne einen externen C-Compiler direkt Objektcode erzeugen. Viele vom C-Compiler erzeugten Warnungen, die durch die C++-Notation hervorgerufen wurden, werden somit verhindert.
Ein wichtiges Merkmal der C++-Compiler ist das strenge Einhalten des **Ansi-C-Standards**.

Aufruf des C++-Compilers, C-Compilers und Linkers:

```
CC  [ optionen ] dateiname.C ...
```

Bei einigen Systemen sieht der Befehl jedoch völlig anders aus (siehe Anhang). Neben den Optionen, die bereits vom C-Compiler bzw. Linker akzeptiert werden, gibt es zusätzlich noch für den C++-Compiler:

- **-Fc** erzeugt C-Quellcode in Datei `dateiname.c`
- **+V** akzeptiert C-Quellcode
- **+L** Zeilennummern werden eingefügt
- **+S** Informationen vom Compiler gehen zur Standardfehlerausgabe

Auch hier gibt es in einigen Systemen Einschränkungen bzw. starke Abweichungen.

2.2 Kommentar

Neben den bekannten Kommentarzeichen **/* ... */** für Kommentare über mehrere Zeilen gibt es in C++ zusätzlich **//** als Kommentarzeichen für Kommentare bis Zeilenende. Beide Zeichen dürfen miteinander verschachtelt werden.

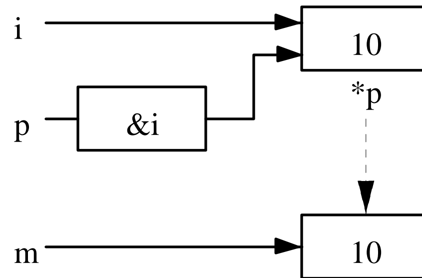
```
/*********************** Zeilenkommentar **********************/
/*                          Kommentar.cc                         */

# include <stdio.h>

main()
{ /* Beginn des C-Kommentars. // C++-Kommentar.
     Ende des C-Kommentars. */

  // C++-Kommentar mit /* C-Kommentar */.

  return 0;
}
```

2.3 Strukturierte Datentypen

2.3.1 Referenzen

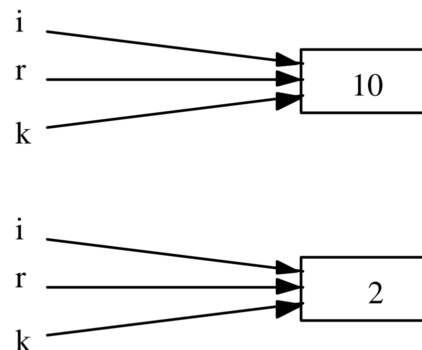
In C adressiert man eine Variable entweder über ihren Namen oder indirekt über einen Zeiger mittels des Adressoperators:

```
/*********************** Zeigeroperator ***********************/
/*                          Zeiger.c                             */

# include <stdio.h>

main()
{ int i = 10;                                          // Name
  int *p = &i;                                        // Zeiger
  int m = *p;                                          // Name

  printf( "%lX: %d\n", &i, i);               // 2FF7FB60: 10
  printf( "%lX: %d\n", p, *p);               // 2FF7FB60: 10
  printf( "%lX: %d\n", &m, m);               // 2FF7FB68: 10

  return 0;
}
```



Durch die Deklaration einer Variablen als **Referenz** auf einen Typ erhält der Adressoperator **&** eine neue Bedeutung. In diesem Falle ist die Referenz ein Alternativname für eine bereits existierende Variable (Original). Folglich muss eine Referenz bei der Deklaration auch definiert werden, diese Initialisierung kann dann nicht mehr verändert werden. Erhält eine Referenz einen neuen Wert, so auch sein Original.

```
/********************** Referenzoperator **********************/
/*                         Referenz.cc                          */

# include <stdio.h>

main()
{ int i = 10;                                                // Name
  int &r = i;                                            // Referenz
  int &k = r;                                            // Referenz

  printf( "%lX: %d\n", &i, i);                       // 2FF7FB58: 10
  printf( "%lX: %d\n", &r, r);                       // 2FF7FB58: 10
  printf( "%lX: %d\n", &k, k);                       // 2FF7FB58: 10

  k = 2;

  printf( "%lX: %d\n", &i, i);                        // 2FF7FB58: 2
  printf( "%lX: %d\n", &r, r);                        // 2FF7FB58: 2
  printf( "%lX: %d\n", &k, k);                        // 2FF7FB58: 2

  return 0;
}
```



Nunmehr bezeichnen **i**, **k** und **r** dieselbe Variable.
Dabei kann für die Deklaration eines Referenztyps **&** analog den Zeigervariablen vor der Referenzvariablen aber auch unmittelbar hinter der Typangabe stehen. Die Schreibweisen `int &p = i;` und `int& p = i;` sind somit gleichbedeutend.

Der eigentliche Sinn dieser Referenztypen wird erst bei der Parameterübertragung in Funktionen deutlich:
Das Programm zum Vertauschen von Variablen

```
/***************** Zeiger als Funktionsparameter ****************/
/*                          Swapping.c                           */
/*                Vertauschen von Variablenwerten                */

#include <stdio.h>

void swap( int *, int *);         // Zeiger als Funktionsparameter

main()
{ int a, b;
  printf( "Eingabe a="); scanf( "%d", &a);
  printf( "Eingabe b="); scanf( "%d", &b);
  printf( "\n");
  swap( &a, &b);                          // Adressen als Argumente
  printf( "Ausgabe a=%d \nAusgabe b=%d.\n", a, b);
  return 0;
}

void swap( int *x, int *y)                   // Funktionsdefinition
{ int temp;
  temp = *x;                          // Zugriff auf Speicherinhalte
  *x = *y;
  *y = temp;
}
```

kann jetzt wie folgt lauten:

```
/************** Referenzen als Funktionsparameter ***************/
/*                         Swapping.cc                           */
/*               Vertauschen von Variablenwerten                 */

#include <stdio.h>
                              // Referenzen als Funktionsparameter
void swap( int &, int &);

main()
{ int a, b;
  printf( "Eingabe a="); scanf( "%d", &a);
  printf( "Eingabe b="); scanf( "%d", &b);
  printf( "\n");
  swap( a, b);                           // Variablen als Argumente
  printf( "Ausgabe a=%d \nAusgabe b=%d.\n", a, b);
  return 0;
}

void swap( int &x, int &y)                   // Funktionsdefinition
{ int temp;
  temp = x;                             // Zugriff auf die Variablen
  x = y;
  y = temp;
}
```

Das zweite Programm hat gegenüber dem ersten den Vorteil, dass das explizite Dereferenzieren mit den *-Operator entfällt. Des weiteren werden bei der Übergabe keine Parameterwerte kopiert und somit weniger Rechenzeit und Speicherplatz benötigt. Interessant ist der Aufruf: **`swap( a, b + 1);`**

Es erfolgt eine Warnung, **x** wird zunächst mit **a** identifiziert und **y** mit einer Hilfsvariablen, in der **b+1** abgelegt wurde. Nach dem Vertauschen hat dann **x** den Wert der Hilfsvariable, also **b+1**, und die Hilfsvariable den Wert von **x**, also **a**. Da man aber auf die Hilfsvariable nach Ablauf der Funktion nicht mehr zugreifen kann, wird zwar letztlich in **a** der Wert von **b+1** stehen, **b** selbst bleibt unverändert.

2.3.2 Strukturen

Für alle Typnamen gibt es jetzt einen gemeinsamen Namensraum, d. h. Namen von Typen einschließlich nutzerdefinierte Typen dürfen nicht parallel mit anderen Typnamen übereinstimmen. Dadurch kann man jetzt bei Variablendeklarationen das Schlüsselwort **struct** weglassen.

```
/************************* Strukturen *************************/
/*                          Punkte.cc                          */
/*             Sortieren von Punkten aus Punktdatei            */

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

struct kartes{ float x, y;};            // Strukturtypdeklaration

struct polar{ float rho, phi;};

struct Punkt
{ kartes kar;                      // struct kann weggelassen werden
  polar pol;
  Punkt *Nf;
};

kartes kartesisch( polar p);
void sortier_ein( Punkt **a, Punkt p);

main()
{ ...
}

/******************** Koordinatenumrechnung *******************/

kartes kartesisch( polar p)
{ ...
}

/*********************** Sortierfunktion **********************/
/*                      von vorn nach hinten                   */

void sortier_ein( Punkt **a, Punkt p)
{ ...
}
```

2.3.3 Verbunde

Bereits in ANSI-C gibt es den Datentype **Verbund** bzw. **Variante**. Während bei Strukturen aufeinanderfolgende Strukturelemente aufeinanderfolgende Speicherplätze belegen, so kann man in Verbunden zu verschiedenen Zeiten den selben Speicher für verschiedene Zwecke benützen.

In C++ kann nun auch für Verbunde, wegen der nicht mehr möglichen Namenskollision, bei den Variablendeklarationen das Schlüsselwort **union** weggelassen werden. Die Verbundvariable kann nunmehr wie folgt deklariert werden.

```
/*************************** Verbund ***************************/
/*                            Verbund.cc                         */

# include <stdio.h>

main()
{ union Verbund
  { int i;
    float f;
    char c;
  };

  Verbund V;                          // union kann weggelassen werden

  V.i = 7;
  printf( "V.i = %d\n", V.i);
  V.f =.5;
  printf( "V.f = %f\n", V.f);

  return 0;
}
```

2.4 Deklarationen

In C++ ist es gestattet, Deklarationen nach Belieben zwischen den Anweisungen zu schreiben. Ihre Wirkung beginnt an der Deklarationsstelle und endet am Ende des Anweisungsblocks. Eine typische Anwendung ist die Deklaration von Laufvariablen in Schleifenkörper:

```
/****************** Beispiel zur For-Anweisung ******************/
/*                            Fak.cc                             */
/*                    Fakultaet ohne Rekursion                   */

# include <stdio.h>

main()
{ int n = 12;
  long Fak = 1;
                        // i innerhalb der for-Anweisung deklariert
  for( int i = 1; i <= n; Fak *= i++);

  printf( "%ld\n", Fak);                              // 479'001'600

  return 0;
}
```

Die Variable **i** gilt bis zum nächsten Blockende.
Bemerkenswert ist die Tatsache, dass man jetzt wieder Felder dynamisch vereinbaren kann. Nach ALGOL und PL1 war das nicht mehr möglich:

```
/************** Dynamische Speicherplatzanforderung *************/
/*                          Z_Vektor.cc                         */
/*         Summation von Vektorkomponenten mit Zeigern          */
/*                     und Zeigerarithmetik                     */

# include <stdio.h>

main()
{ int LAENGE;
```

```
    printf( "Laenge des Vektors:");
    scanf( "%d", &LAENGE);

    float v[ LAENGE];         // dynamische Speicherplatzanforderung
                                         // Einlesen der Komponenten

    printf( "Eingabe des Vektor: \n");
    for( int i=0; i<LAENGE; i++) scanf( "%f", v+i);

    float Summe=0;                     // Summation der Komponenten

    for ( float *z=v; z<v+LAENGE; z++)  Summe+=*z;

    printf( "Die Summe der Komponenten ist %.2f.\n", Summe);

    return 0;
  }
```

2.5 Funktionen

In C++ müssen Funktionen wie in ANSI-C vor ihrem ersten Aufruf deklariert sein. Darum sind viele ältere C-Programme mit C++ nicht kompatibel.

2.5.1 Vorbesetzte Parameter

Für Funktionsparameter können Vorbesetzungen (Default-Werte) vorgegeben werden. Diese werden beim Funktionsaufruf verwendet, wenn keine oder nicht alle Argumentwerte angegeben wurden. Vorbesetzte Parameter dürfen nur von rechts nach links in der Parameterliste angegeben werden, da sonst die Eindeutigkeit der Zuordnung nicht gewährleistet ist.

```
/********************* Vorbesetzte Parameter *******************/
/*                             Parameter.cc                       */
/*                Summation von 4 Integer-Groessen                */

# include <stdio.h>

// Funktionsdeklaration mit vorbesetzten Parametern
int add( int i, int j, int m = 0, int n = 0);

main()
{ printf("add(1,2,3,4) = %d\n", add(1,2,3,4));               // 10
  printf( "add(1,2,3) = %d\n", add( 1,2,3));                  // 6
  printf( "add(1,2) = %d\n", add(1,2));                       // 3
  // printf( "add(1) = %d\n", add(1));                      FEHLER!
  // printf( "add(1,2,,4) = %d\n", add(1,2,,4));            FEHLER!

  return 0;
}

// Funktionsdefinition ohne Vorbesetzung in der Kopfzeile
int add( int i, int j, int m, int n)
{ return i + j + m + n;
}

// int add( int i = 0, int j, int m, int n = 0);            FEHLER!
```

2.5.2 inline-Funktionen

Zur Einsparung von Laufzeit und Größe des Objektcodes können kleinere Funktionen als **inline** deklariert werden. Bei der Compilierung wird vom Compiler eine inline-

Funktion ähnlich eines Macros behandelt, d. h. an den Aufrufstellen wird eine Zeilenexpansion durchgeführt. Die Semantik der Funktion bleibt erhalten, die bei Makros oft auftretenden Fehler werden vermieden. Ist der Funktionsumfang zu groß oder die Funktion rekursiv, so wird die Funktion wie eine Funktion ohne inline behandelt.

```
/********************** Inline-Funktionen *********************/
/*                         Inline.cc                          */
/*              Summation von 4 Integer-Groessen              */

# include <stdio.h>

// inline - Funktionsdeklaration und Funktionsdefinition
inline int add( int i, int j, int m = 0, int n = 0)
{ return i + j + m + n;
}

main()
{ printf( "add(1,2,3,4) = %d\n", add(1,2,3,4));            // 10
  printf( "add(1,2,3) = %d\n", add(1,2,3));                  // 6
  printf( "add(1,2) = %d\n", add(1,2));                      // 3

  return 0;
}
```

`x = add(1,2);` expandiert in **`x = ( 1 + 2 + 0 + 0);`**

Je größer die inline-Funktion, desto geringer ist der relative Zeitvorteil durch den Wegfall der Parameterübergabe usw. usf., und desto größer der Nachteil der Code-Aufblähung. Deshalb ist die Anwendung nur für kurze Funktionen, die nur aus wenigen Anweisungen bestehen, zu empfehlen.

2.5.3 Überladen von Funktionen

C++ führt eine strenge Typüberprüfung durch. Wird z. B. eine Funktion mit Parametern vom Typ **int** erwartet und beim Funktionsaufruf aber Argumente vom Typ **float** angegeben, so meldet der Compiler einen Fehler.
Man kann nun ggf. verschiedene Versionen dieser Funktionen definieren, jede mit anderen Parametertypen. Obwohl alle im Prinzip das gleiche tun, müsste man nach der C-Konvention jeder einen anderen Namen geben. Das lässt sich nun in C++ vermeiden, indem man die Funktion **überlädt**. Mehrere Funktionsversionen erhalten dabei den gleichen Namen, unterscheiden sich aber durch verschiedene Parameterlisten (Die Reihenfolge und Typen der Parameter nennt man **Signatur** der Funktion, der Funktionstyp selber gehört nicht dazu.). Der Compiler kann nun an Hand der aktuellen Parameter unterscheiden, welche Version auszuführen ist. Bei der Verwendung von vorbesetzten Parametern können u. U. Mehrdeutigkeiten auftreten, diese sollte man tunlichst vermeiden, da dann das Verhalten des Compilers implementationsabhängig wird.

```
/***************** Ueberladen von Funktionen *****************/
/*                          Swap.cc                           */
/*              Vertauschen von Variablenwerten               */

#include <stdio.h>
                                        // int-Funktionsdefinition
inline void swap( int &x, int &y)
```

```
{ int temp; temp = x; x = y; y = temp;
}
                                        // float-Funktionsdefinition
inline void swap( float &x, float &y)
{ float temp; temp = x; x = y; y = temp;
}

main()
{ int a, b;
  printf( "int-Eingabe a="); scanf( "%d", &a);
  printf( "int-Eingabe b="); scanf( "%d", &b);
  printf( "\n");
  swap( a, b);                                          // int-swap
  printf( "Ausgabe a=%d \nAusgabe b=%d.\n", a, b);

  float r, s;
  printf( "float-Eingabe r="); scanf( "%f", &r);
  printf( "float-Eingabe s="); scanf( "%f", &s);
  printf( "\n");
  swap( r, s);                                        // float-swap
  printf( "Ausgabe r=%f \nAusgabe s=%f.\n", r, s);

  // swap( a, s);                                       // FEHLER !
  // printf( "Ausgabe a=%d \nAusgabe s=%f.\n", a, s);

  return 0;
}
```

2.6 Operatoren

2.6.1 Speicherverwaltung

Der Benutzer hat durch die C++-Operatoren **new** und **delete** der Prioritätsstufe 14 die Möglichkeit, die Speicherplatzverwaltung selber in die Hand zu nehmen. Beide Operatoren verwenden intern die von C her bekannten Funktionen **alloc**, **malloc** und **free**.

Zur dynamischen Bereitstellen von Speicher (**Allocation**) wird in C++ vorzugsweise der Operator **new** verwandt. Die Ausdrücke **new *typ*** bzw. **new(*typ*)** reservieren hinreichend viel Speicher für Daten vom Typ ***typ*** und liefern einen Zeiger vom Typ ***typ**** auf diesen Speicher zurück. Kann kein Speicher angefordert werden, so wird der Nullzeiger zurückgeliefert.

In Klammern kann hinter dem Ausdruck ein Initialisierungswert angegeben werden, sonst ist der Wert unbestimmt.

Der Speicher wird freigegeben (**Deallocation**), indem man den Operator **delete** auf den Zeigerwert anwendet. Mit **delete adresse** bzw. **delete[anzahl] adresse** wird ein durch **new** erzeugter Verweis wieder aufgehoben, wobei evtl. **anzahl** die Zahl der zu löschenden Verweise angibt.

```
/********************* Speicherverwaltung **********************/
/*                          Speicher.cc                          */

# include <stdio.h>
# include <string.h>

main()
{ char *Text = new char[ 100];              // Speicherreservierung
  int *Jahr = new int( 1995);                // mit Initialisierung

  strcpy( Text, "Hallo, Welt des Jahres ");
  printf( "%s%d!\n", Text, *Jahr);
```

```
    delete Jahr;                    // löscht reservierten Speicher
    delete Text;
            // delete[ 100] Text; erzeugt u.U. Warnung, sonst analog

    return 0;
  }
```

2.6.2 Der Zugriffsoperator ::

Hat eine lokale Variable den gleichen Namen wie eine globale Variable, so ist die globale Variable innerhalb des Blockes, in dem die lokale Variable deklariert wurde, verschattet, d. h. man kann auf sie nicht zugreifen. In C++ gibt es durch den Operator `::` vor dem Variablennamen die Möglichkeit, explizit auf die **globale** Variable zuzugreifen.

```
/*********************** Zugriffsoperator ***********************/
/*                          Zugriff.cc                            */

# include <stdio.h>

int xyz = 1;

main()
{ float xyz = .5;

  printf( "global: %d\n", ++::xyz);                              // 2
  printf( "lokal:  %.2f\n", xyz *= .5);                       // 0.25

  return 0;
}
```

Der Zugriffsoperator (Scope Resolution Operator) `::` hat die Prioritätsstufe 15 (höchste Prioritätsstufe), kann aber nur auf globale Variablen angewandt werden.