C++ Einführung

Dipl.-Medieninform.(FH) Severin S. Todt, MSc.

Fachgruppe Computergraphik und Multimediasysteme

Universität Siegen – Fachbereich 12

Version : 04.11.2003

Übersicht :

Übersicht (forts.):

5. Strukturierte Datentypen

- 5.1. Arrays
- 5.2. Structs
- 5.3. Unions
- 5.4. Enumerations
- 5.5. Pointer
  - 5.5.1. Deklaration von Pointern
  - 5.5.2. Pointer Operationen
  - 5.5.3. Zeigerarithmetik
  - 5.5.4. Dynamische Datenstrukturen mit Zeigern
  - 5.5.5. Garbage Collection

4. Kontrollstrukturen

4.1. Blöcke

- Blöcke umfassen eine beliebige Menge von Anweisungen
- Ein Block ist begrenzt durch geschweifte Klammern
- Ein Block kann den Sichtbarkeitsbereich einer Variablen einschränken

4.2. `if` Statement

- Das `if` Statement implementiert eine bedingte Anweisung, auch Alternative oder Verzweigung genannt

if(<bool Expression>)
  <Anweisung>
[else <if Statement>]
  <Anweisung>
[else]
  <Anweisung>

- Abhängig von der Auswertung des im `if` Statement angegebenen boolschen Ausdrucks wird ein definierter Anweisungsblock verarbeitet

```
int value(3);
if (value==1)
  cout << “sehr gut” << endl;
else if (value==2)
  cout << “gut” << endl;
else if (value==3)
  cout << “befriedigend” << endl;
else if (value==4)
  cout << “ausreichend” << endl;
else if (value==5)
  cout << “mangelhaft” << endl;
else
  cout << “ungenügend” << endl;
```

→ *befriedigend*

4.3. `switch` – Verteiler

- Ein `switch` – Verteiler setzt eine Mehrwegauswahl um
- Beim `switch` – Verteiler wird abhängig von einem Ausdruck bestimmt, welche Alternative zu verarbeiten ist
- Die Auswahl der zu verarbeitenden Alternative erfolgt über den Abgleich des im `switch` gegebenen Ausdrucks mit denen für jedes Target individuell definiertem konstanten Ausdruck

```
switch (<ausdruck>)
{
   case <const ausdruck>:
        <Anweisung>
        [break;]
   [default:
        <Anweisung>
        [break;] ]
}
```

```
int note(3);
switch(note)
{
 case 1:
   cout << "sehr gut" << endl;
   break;
 case 2:
   cout << "gut" << endl;
   break;
 case 3:
   cout << "befriedigend" << endl;
   break;
 case 4:
   cout << "ausreichend" << endl;
   break;
 case 5:
   cout << "mangelhaft" << endl;
   break;
 default:
   cout << "ungenügend" << endl;
   break;
}
```

→ *befriedigend*

```
int note(3);
switch(note)
{
 case 1:
   cout << "sehr gut" << endl;
 case 2:
   cout << "gut" << endl;
 case 3:
   cout << "befriedigend" << endl;
 case 4:
   cout << "ausreichend" << endl;
 case 5:
   cout << "mangelhaft" << endl;
 default:
   cout << "ungenügend" << endl;
}
```

→ *befriedigend*
ausreichend
mangelhaft
ungenügend

4.4. `for` – Schleife

- Die `for` – Schleife wiederholt eine Anweisung solange die im Kontext der Schleife angegebene Bedingung erfüllt bleibt
- Durch die Konstrukte der Initialisierung und der Inkrementierung sind sie besonders geeignet für Schleifen mit determinierbarer Durchlaufanzahl

for (<Initialisierung>; <boolsche Bedingung>; <Inkrement Anweisung>)
<Anweisungsblock>

```
for (int i=10;i>0; i--)
  cout << i << ", ";
cout << "FIRE" << endl;
```

→ *10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE*

4.5. `while` / `do … while` Schleife

- Die `while` – Schleife wiederholt eine Anweisung solange die im Kontext der Schleife angegebene Bedingung erfüllt bleibt
- Die Durchlaufbedingung wird bei Schleifeneintritt geprüft

while (<boolsche Bedingung>)
<Anweisungsblock>

```
int count=10;
while(count > 0)
  cout << count-- << ", ";
cout "FIRE" endl;
```

→ *10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE*

- Die `do … while` – Schleife wird mindestens einmal durchlaufen, da erst nach dem Durchlauf die Bedingung geprüft wird

do
* <Anweisungsblock>*
while (<boolsche Bedingung>);

```
int count=10;
do
  cout << count-- << ", ";
while(count > 0);
cout "FIRE" endl;
```

→ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE

4.6. Sprunganweisungen

- Sprunganweisungen dienen dem gezielten Sprung an andere Codesegment – Stellen während der Programmausführung
- Sprunganweisungen können dazu verwendet werden, Kontrollstrukturen zu beliebigen Ausführungszeitpunkten zu verlassen
- Sprunganweisungen können bei unsachgemäßen Gebrauch undefinierte Zustände hinterlassen → sind mit Sorgfalt zu verwenden

4.6.1. `break`

- Das **`break`** Statement kann in Schleifenkonstrukten und **`switch`** – Anweisungen zum Einsatz kommen
- **`break`** ermöglicht, ein Kontrollkonstrukt zu verlassen, auch wenn die Bedingung zum Verlassen noch nicht erfüllt ist
  - Bei Schleifenkonstrukten, obwohl die Schleifenbedingung erfüllt ist
  - Bei **`case`** – Verteilern, noch bevor alle Fälle abgearbeitet sind

4.6.2. `continue`

- Das **`continue`** Statement kann in Schleifenkonstrukten zum Einsatz kommen
- **`continue`** bewirkt einen Sprung innerhalb eines Schleifenkonstruktes direkt an das Ende des Anweisungsblockes der Schleife und bewirkt so ein sofortiges erneutes Auswerten der Abbruchbedingung

```
for(int i=0;i<=10;i++) {
  if(i==5)
    continue;
  else
    cout << i << " ";
}
```

→*0 1 2 3 4 6 7 8 9 10*

4.6.3. `goto`

- **`goto`** bewirkt einen absoluten Sprung innerhalb des Programms bis hin zu einer definierten Sprungmarke
- Absolute Sprünge sind mit Vorsicht zu verwenden, da Randbedingungen bei dieser Art von Sprüngen nicht berücksichtigt werden, d.h. es könnten inkonsistente Bedingungen erzeugt werden
- Eine Sprungmarke wird über einen eindeutigen Bezeichner und einen schließenden Doppelpunkt gekennzeichnet

label::= <identifier> :

goto <label>;

```
int n(10);
loop:
cout << n-- << ", ";
if(n>0) goto loop;
cout << "FIRE" << endl;
```

→ *10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE*

4.6.4. `return`

- Das **`return`** Statement bewirkt einen Rücksprung aus einer Routine hin zu dem Punkt, an dem sie aufgerufen wurde
- Je nach Deklaration der Routine aus der zurückgesprungen wird, ist mit der **`return`** Anweisung ein Wert zurückzugeben, der dem Typ der Funktionsdeklaration entspricht

```
double eval5( ){
  return 5.0;
}

int main(int argc, char** argv){
  for(int i=10;i>0;i--){
    if(i!=5)
      cout << i << ", ";
    else
      cout << eval5() << ", ";
  }
  cout << "FIRE" << endl;
}
```

→ 10, 9, 8, 7, 6, 5.0, 4, 3, 2, 1, FIRE

5. Strukturierte Datentypen

5.1. Arrays

- Arrays bezeichnen eine Folge von Elementen (Variablen) gleichen Typs, die sequentiell hintereinander im Speicher abgelegt sind
- Die Deklaration eines Arrays erfolgt unter Angabe des Basistypen und der zur Compilezeit feststehenden Größe des Datenfeldes
- Die Elemente eines Arrays können durch Angabe eines zusätzlichen Indexes zu einem Variablenbezeichner referenziert werden (Der Index eines Arrays startet immer bei 0)

<type> identifier [size]

```
#define SIZE (3)
int  integerFeld[SIZE];
integerFeld[0]=2;  integerFeld[1]=4;  integerFeld[2]=6;

cout << "Das Feld ist " << SIZE << " Felder groß."
        "Der letzte Eintrag ist "<< integerFeld[2] << endl;
```

→ Das Feld ist 3 Felder groß. Der letzte Eintrag ist 6

5.1.1. Initialisierung von Arrays

- Abhängig von der Sichtbarkeit eines deklarierten Arrays, kann unter Umständen eine automatische Initialisierung erfolgen
- Bei global deklarierten Arrays werden die einzelnen Elemente des Arrays mit "0" initialisiert
- Lokal deklarierte Arrays werden nicht initialisiert, d.h. der Inhalt der einzelnen Elemente ist nicht definiert
- Eine explizite Initialisierung von Arrays erfolgt durch Angabe der Menge von Werten für die einzelnen Elemente in geschweiften Klammern

  <type> identifier [size]={valueOfType,…,valueOfType}

- Alternativ kann bei einer Initialisierung des Feldes die Angabe der Größe weggelassen werden, dann ergibt sich die Feldgröße aus der Anzahl von Initialisierungswerten

5.6. Void Pointer

- Neben den dynamischen Datenstrukturen lässt sich mehr Flelxibilität erreichen durch die Einführung von **`void`** Pointern
- **`void`** Pointer können auf beliebige Datentypen zeigen
- Erst zur Laufzeit wird mit der Speicherallozierung und Adresszuweisung ein Typ angegeben auf den der Pointer zeigt
- Der Datentyp der durch einen **`void`** Pointers referenziert werden kann, kann zur Laufzeit variieren
- Die Angabe des aktuellen Typs ist bei der Dereferenzierung zum Zweck der Pointer Arithmetik zwingend notwendig
- Die Typisierung erfolgt durch ein explizites Casting

```
void* pointer = new int[5];
*(((int*)pointer)+0)=0;
*(((int*)pointer)+1)=1;
*(((int*)pointer)+2)=2;
*(((int*)pointer)+3)=3;
*(((int*)pointer)+4)=4;
cout << *(((int*)pointer)+4) << endl;
```

→4

5.1.2. Mehrdimensionale Arrays

- Arrays sind nicht auf eine Dimension begrenzt
- Die Einführung mehrdimensionaler Arrays ermöglicht einen komfortableren Zugriff auf die interne Repräsentation
- Ein zweidimensionales 2x3 großes Array bleibt intern als eindimensionales 6 Felder großes Array implementiert
- Die Deklaration erfolgt unter Angabe der Größe der einzelnen Dimensionen

<type> identifier[size1]…[sizeN]

```
int points[2][3];
```

- Wie auch bei eindimensionalen Arrays erfolgt bei globaler Deklaration eine Initialisierung der Felder mit "0"
- Explizit kann eine Initialisierung erfolgen, indem jede Dimension des Arrays in der darüberliegenden Dimension mit geschweiften Klammern geschachtelt initialisiert wird

```
int points[2][3]={{1,2,3},{4,5,6}};
```

- Auch bei mehrdimensionalen Arrays kann auf Größenangaben verzichtet werden, jedoch nur bei der Angabe der ersten Dimension; die Größe der abhängigen Felder ist zur internen Verarbeitung zwingend anzugeben

```
int points[][3]={{1,2,3},{4,5,6}};
```

5.1.3. Charakter Arrays oder Strings

- Strings stellen eine besondere Form von Arrays dar
- Strings werden intern als Array von Charakter Werten repräsentiert
- Ein String besteht aus der Menge ihn beschreibenden Charakter und eines zusätzlichen terminierenden Nullzeichens "\0" , um das Ende des Strings zu kennzeichnen
- Das einen String beschreibende Array ist somit immer 1 Feld größer als der String an sichtbaren Zeichen besitzt

```
char hallo[6]={'h', 'a', 'l', 'l', 'o', '\0'};
cout << hallo << endl;
```

→ *hallo*

- Bei der Initialisierung von Charakter Arrays kann ein String in doppelten Anführungszeichen ohne Angabe der Array Größe genutzt werden; das resultierende Feld hat dann eine Größe von Stringlänge + 1

```
char myName = "Homer";
```

5.2. Structs

- Structs gruppieren einen Satz möglicherweise unterschiedlicher Datentypen und deklariert diese unter einem gemeinsamen Namen
- Der für das Struct gewählte gemeinsame Struct-Name, kann für weitere Variablendeklarationen Verwendung finden
- Wird kein Struct-Name angegeben kann die Deklaration nur für den im direkten Anschluß angegebenen Variablenbezeichner Verwendung finden

```
struct [struct_name] {
    <type>  elementName1;
    …
    <type>  elementNameN;
}[identifier];
```

- Der Zugriff auf die einzelnen Komponenten des Structs erfolgt über den Punkt Operator (".")
- Structs können auch geschachtelt verwendet werden

```
#include <string.h>
#include <iostream>

struct Person {
  char name[128];
  int   alter;
  float gewicht;
} homer;
Person marge;

int main(int argc, char** argv){
  homer.name[0] = 'H';
  homer.name[1] = 'o';
  homer.name[2] = 'm';
  homer.name[3] = 'e';
  homer.name[4] = 'r';
  homer.name[5] = '\0';
  homer.alter = 36;
  homer.gewicht = 260.00;

  strcpy(marge.name,"Marge Bouvier Simpson");
  marge.alter = 34;
  marge.gewicht = 130.00;
```

```
  cout << "Die Eltern von Bart, Lisa und Maggy sind : " << endl;
  cout << "Vater: " << homer.name << ", " << homer.alter <<" Jahre, "
       << homer.gewicht << "lbs" << endl;
  cout << "Mutter: " <<marge.name << ", " << marge.alter <<" Jahre, "
       << marge.gewicht << "lbs" << endl;

  return 0;
}
```

→ *Die Eltern von Bart, Lisa und Maggy sind :*
Vater: Homer, 36 Jahre, 260 lbs
Mutter: Marge Bouvier Simpson, 34 Jahre, 130 lbs

5.3. Unions

- Unions ermöglichen die Vereinigung von Komponenten verschiedenen Typs unter einem Union-Namen
- Ein Union belegt dabei soviel Speicherplatz wie der größte im Union deklarierte Datentyp

```
union [union_name] {
    <type>  elementName1;
    …
    <type>  elementNameN;
}[identifier];
```

- Der für das Union gewählte gemeinsame Union-Name, kann für weitere Variablendeklarationen Verwendung finden
- Wird kein Union-Name angegeben kann die Deklaration nur für den im direkten Anschluß angegebenen Variablenbezeichner Verwendung finden
- Der Zugriff auf die einzelnen Komponenten des Union erfolgt über den Punkt Operator (".")

```
union Operand {
  int i;
  float f;
};
Operand op1,op2;

op1.i=4;
op2.f=4.0;

cout << "6/4 = " << 6/op1.i << endl;
cout << "6/4 = " << 6/op2.f << endl;
```

→ *6/4 = 1*
6/4 = 1.5

5.4. Enumerations

- Enumeration bezeichnet einen Aufzählungstyp mit frei wählbarem Wertebereich
- Zur Deklaration einer Enumeration werden alle für diesen zu deklarierenden Grundtyp möglichen Werte in geschweiften Klammern angegeben

  enum [enumeration_name]
  {value1, … , valueN}[idenitifer];

- Der für die Enumeration gewählte Enumeration-Name, kann für weitere Variablendeklarationen Verwendung finden
- Wird kein Enumeration-Name angegeben kann die Deklaration nur für den im direkten Anschluß angegebenen Variablenbezeichner Verwendung finden

```
enum tag
  {montag, dienstag, mittwoch, donnerstag, freitag, samstag, sonntag}
```

- Intern werden die angegebenen Werte der Enumeration auch als Integer representiert (beginnend bei 0 erhalten die nachfolgend definierten Werte einen um 1 erhöhten Integer Wert)

```
enum tag
  {montag=0,dienstag=1,mittwoch=2,donnerstag=3,freitag=4,samstag=5,
   sonntag=6}
```

- Anstatt der impliziten internen Integer Repräsentationen, können explizit Integer Repräsentationen der Werte angegeben werden

```
enum tag
  {montag=1,dienstag=2,mittwoch=3,donnerstag=4,freitag=5,samstag=6,
   sonntag=7}
```

- Die Integer Repräsentation ermöglicht dann die Auswertung von Vergleichsoperatoren

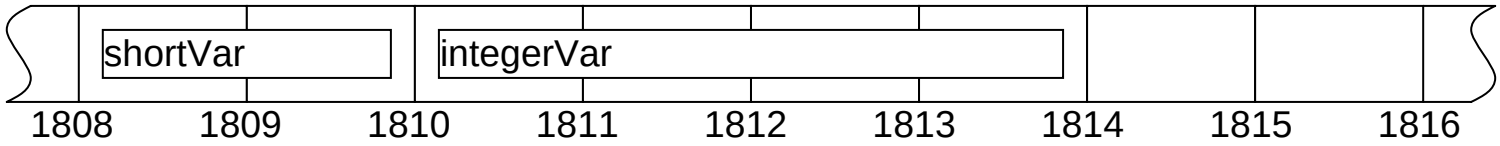
```
enum tag
 {montag=1,dienstag=2,mittwoch=3,donnerstag=4,freitag=5,samstag=6,
  sonntag=7}

int main(int argc, char** argv){
  tag day=dienstag;
  if(day<=samstag)
     cout << "Arbeitstag" << endl;
  else
     cout << "Wochenende" << endl;
 }
```

→ *Arbeitstag*

5.5. Pointer

- Bei der Ausführung einer Anwendung, werden alle definierten Variablen im Speicher abgelegt
- Der Zugriff auf den Inhalt der Variablen erfolgt intern durch die Verarbeitung der Inhalte von Speicherzellen
- Intern ist es notwendig, mit jeder Variable die entsprechende Speicherzelle zu assoziieren
- Die intern verfügbare Adresse einer Speicherzelle kann auch in der Anwendungsentwicklung genutzt werden
- Bei der Anwendungsentwicklung existieren Pointer Typen, die in der Lage sind, Referenzen auf einen beliebigen Speicherbereich aufzunehmen

```
int integerVar;
short shortVar;
```

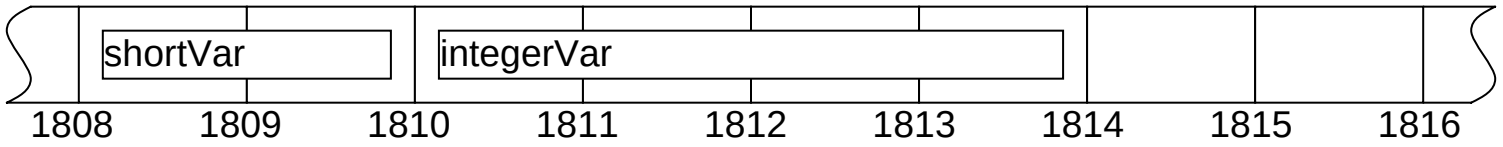


5.5.1. Speicheradressen von Variablen

- Die Adresse der Speicherzelle, in der der Wert einer Variable gespeichert ist, kann über den "&" Operator abgerufen werden
- Der Adressoperator wird auch als Dereferenzoperator bezeichnet

```
int integerVar=5;

cout << &integerVar << endl;
```

→ 1810

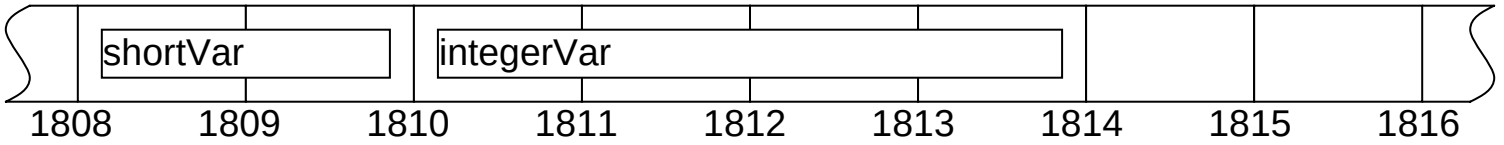


5.5.2. Werte von Speicheradressen

- Neben dem Adressoperator existiert weiter ein Referenzoperator mit Hilfe dessen es möglich ist, den Inhalt einer Speicherzelle zu referenzieren
- Um den Inhalt einer Speicherzelle referenzieren zu können, ist die Adresse der Speicherzelle zu verwenden
- Der Referenzoperator ist als "*" gegeben

```
int integerVar=5;

cout << *(&integerVar) << endl;
```

→ 5



5.5.3. Deklaration von Pointern

- Um die Adresse einer Speicherzelle wiederum in einer Variablen halten zu können, bedarf es eines Pointer Typs
- Pointer sind in der Lage, Speicheradressen aufzunehmen
- Aufgrund der Tatsache, dass es mit Pointern möglich ist, Werte aus einer Speicherzelle herauslesen zu können, muß bekannt sein, welcher Datentyp durch den Pointer referenziert wird

  *<type> * pointer_name;*

- Die Adresse, die der Pointer referenziert erhält man durch den Dereferenzoperator "&"
- Zu dem Wert einer Speicheradresse erhält man Zugang durch den Referenzoperator "*"

```
int * intPoint;
int intValue=3;

intPoint = &intValue;
cout << *intPoint << " – " << intValue << endl;
```

→3 - 3

```
intValue=0;
intValue=*intPoint;
cout << *intPoint << " – " << intValue << endl;
```

→0 - 0

```
*intPoint = 1;
cout << *intPoint << " – " << intValue << endl;
```

→1 - 1

5.5.4. Initialisierung von Pointern

- Wie bei Variablen üblich, können auch Pointer bei der Deklaration initialisiert werden
- Die Initialisierung erfolgt durch Angabe der Adresse einer anderen Variable oder aber der definierten Konstante "NULL“
- "NULL" wird als Null-Pointer bezeichnet und steht für die Adresse 0, also eine nicht existente Adresse

```
int intValue=3;
int * intPoint1=&intValue;
int * intPoint2=NULL;
```

- Jeder Versuch, zur Laufzeit auf eine ungültige Speicheradresse zuzugreifen, löst einen so genannten "Segmentation fault" also einen Speichersegmentierungsfehler aus

```
int * intPoint2=NULL;
cout << *intPoint2 << endl;
```

→ *Segmentation fault*

5.5.5. Pointer und Arrays

- Die Deklaration eines Arrays der Größe n beschreibt nichts anderes, als einen Speicherbereich der in der Lage ist, n Elemente des deklarierten Typs aufzunehmen
- Die Array Variable beinhaltet nach der Deklaration die Adresse des ersten Elementes im Feld und ist somit ein Pointer

```
int intArray[]={1,2,3,4,5};
int * intPoint=intArray;
cout << *intPoint << endl;
```

→1

5.5.6. Pointer Arithmetik

- Bei einem definiertem Array erhält man Zugriff auf den Inhalt des n-ten Elements durch Indizierung

```
int intArray[]={1,2,3,4,5};
cout << intArray[1] << endl;
```

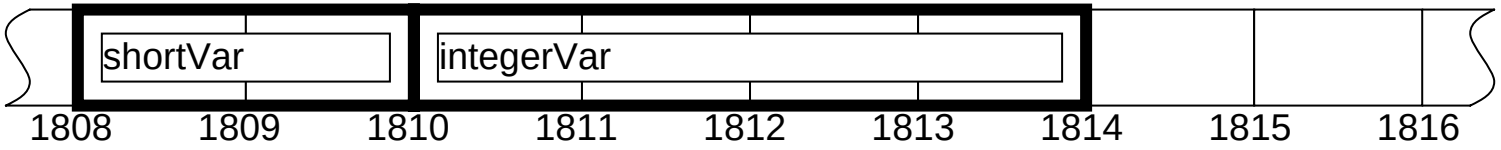
→*2*

- Wenn ein Array ein Pointer ist, und über Indizierung ein Zugriff auf nachfolgende Elemente erfolgen kann, so muß der Zugriff auf das n-te Element auch über die Verwendung eines Pointers möglich sein

```
int intArray[]={1,2,3,4,5};
int * intPoint=&intArray[1];
cout << *intPoint << endl;
```

→*2*

- Das n-te Element eines Arrays zu referenzieren, heißt nichts anderes, als dasjenige Element zu referenzieren, das vom ersten Element im Speicher n Schritte entfernt liegt
- Die Schrittweite ergibt sich dabei aus dem Basisdatentyps des Arrays, bzw. des Pointers



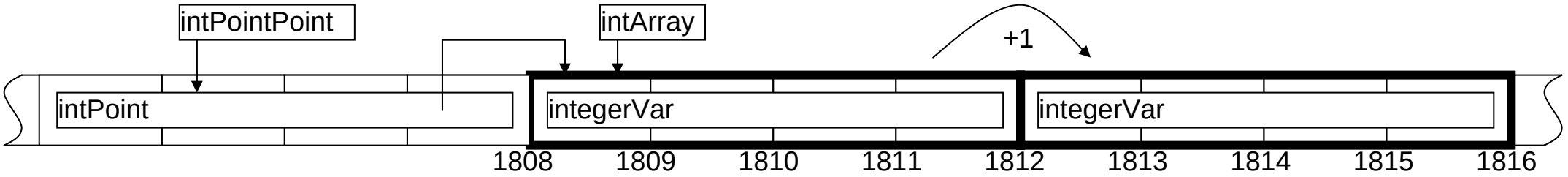
- Ist der Startpunkt des Arrays als Adresse im Speicher bekannt, so ist ausgehend davon der Zugriff auf nachfolgende Elemente, durch einfache arithmetische Operationen möglich

```
int intArray[]={1,2,3,4,5};
int * intPoint=intArray;
cout << *(intPoint+0) << "=" << intArray[0] << endl;
cout << *(intPoint+1) << "=" << intArray[1] << endl;
*(intPoint+2)=7;
cout << *(intPoint+3-1) << "=" << intArray[2] << endl;
```

→ *1=1*
2=2
7=7

5.5.7. Pointer auf Pointer

- Völlig legal ist die Deklaration eines Pointers auf einen anderen Pointer, d.h. ein Pointer der auf einen Speicherbereich verweist, in der die Adresse einer anderen Speicherzelle gespeichert ist




```
int intArray[]={1,2,3,4,5};
int * intPoint=intArray;
int ** intPointPoint=&intPoint;
cout<<*(*intPointPoint)<<"="<<*intPoint<<"="<<intArray[0]<<endl;
cout<<*((*intPointPoint)+1)<<"="<<*(intPoint+1)<<"="
    <<intArray[1]<<endl;
```

→ 1=1=1
2=2=2

5.5.8. Dynamische Datenstrukturen mit Zeigern

- Wenn der direkte Zugriff auf die Adresse und den Wert einer Variablen über die Variable selbst erfolgen kann, wo liegt der eigentliche nutzen von Pointern
- Neben der effizienten Handhabe von Pointern ermöglichen Pointer die Arbeit mit dynamischen Datenstrukturen
- Dynamische Datenstrukturen sind Strukturen, die erst zur Laufzeit erzeugt, vergrößert, verkleinert und zerstört werden
- Die Größe dynamischer Strukturen steht nicht wie bei allen vorher besprochenen Deklarationen schon zur Compilezeit fest
- Um dynamisch Datenstrukturen zu erzeugen, sind zuerst Speicherzellen anzufordern die in der Lage sind, eine bestimmte Anzahl n eines bestimmten Datentyps aufzunehmen; dies geschieht mit Hilfe des **new** Operators
- Der **new** operator liefert eine Adresse auf den reservierten Speicherbereich zurück

new <type>
oder
new <type>[n]

```
int* pointer;
pointer = new int;
```

- Ist der Speicher alloziert, so kann dieser beliebig genutzt werden
- Wird der Speicherbereich nicht weiter benötigt, ist dieser IMMER wieder freizugeben
- Zum Freigeben von Speicher steht als Gegenspieler zum **`new`** Operator der **`delete`** Operator zur Verfügung

  delete identifier
  oder
  delete[] identifier

- Merke : Zu jedem Vorkommen des **`new`** Operators im Quellcode sollte ein **`delete`** Operator vorkommen
- Dynamisch allozierter Speicher wird nicht automatisch wieder freigegeben

5.6. Void Pointer

- Neben den dynamischen Datenstrukturen lässt sich mehr Flexibilität erreichen durch die Einführung von **`void`** Pointern
- **`void`** Pointer können auf beliebige Datentypen zeigen
- Erst zur Laufzeit wird mit der Speicherallozierung und Adresszuweisung ein Typ angegeben auf den der Pointer zeigt
- Der Datentyp der durch einen **`void`** Pointers referenziert werden kann, kann zur Laufzeit variieren
- Die Angabe des aktuellen Typs ist bei der Dereferenzierung zum Zweck der Pointer Arithmetik zwingend notwendig
- Die Typisierung erfolgt durch ein explizites Casting

```
void* pointer = new int[5];
*(((int*)pointer)+0)=0;
*(((int*)pointer)+1)=1;
*(((int*)pointer)+2)=2;
*(((int*)pointer)+3)=3;
*(((int*)pointer)+4)=4;
cout << *(((int*)pointer)+4) << endl;
```

→4