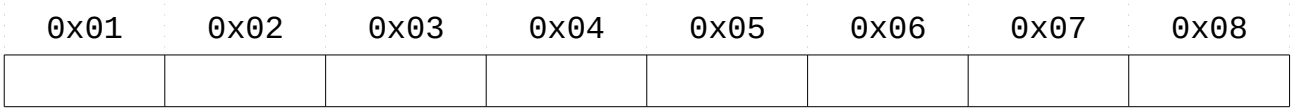
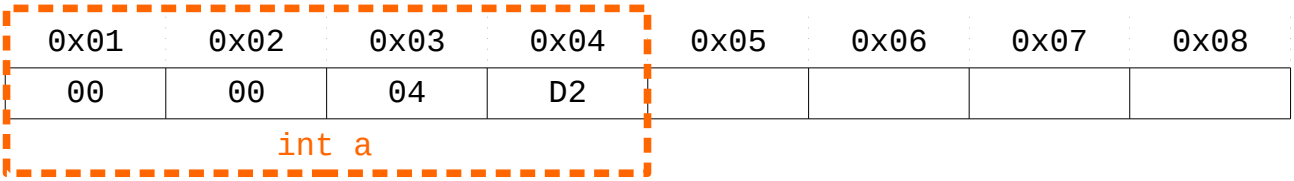
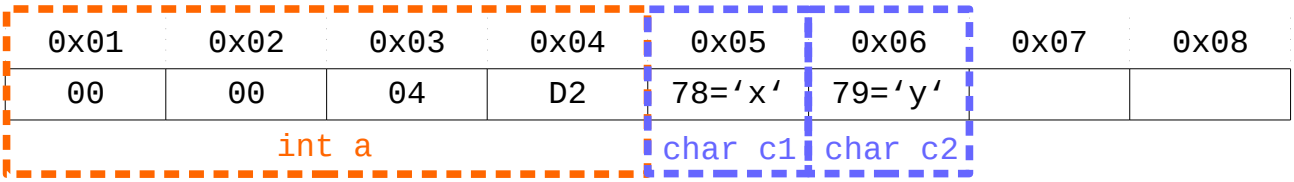
Im Folgenden möchte ich die letzte verbleibende Art von Typen der Programmiersprache C einführen – die Pointer. Um das Prinzip und den Nutzen von Pointern zu verstehen, wollen wir uns auf einer abstrakten Ebene anschauen, wie der Speicher in einem einfachen Programm aussieht. Dazu möchte ich mich sehr weit von der echten Implementierung distanzieren um ein möglichst einfaches Minimalbeispiel zu konstruieren. Wir stellen zunächst den Speicher als ein fortlaufendes Feld von Bytes dar. Jedes Byte hat eine eigene Adresse, welche ich in der Kopfzeile vermerke, darunter der Inhalt der Zelle.



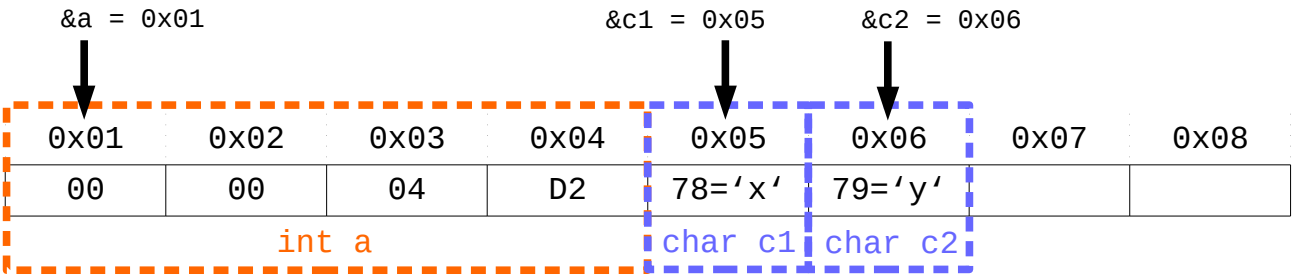
Zur weiteren Vereinfachung nehmem wir an, dass unsere Architektur Werte im Big-Endian-Format speichert, also fortlaufend vom höchstwertigen Byte zum niederwertigsten Byte. Wir stellen ein Byte als eine Hex-Zahl mit 2 Ziffern dar (kombinatorische Herleitung für die Darstellungsform sollte klar sein). Ein Beispiel für die Big-Endian-Darstellung: Wir wollen die Zahl 1234_{10} in einer 4 Byte langen Integer-Variablen speichern. Dazu führen wir einen Basiswechsel von 10 zu 16 durch und speichern die Ziffern fortlaufend. $1234_{10} = 4D2_{16}$. Nun belegen wir unsere 4 Byte wie folgt: `00 00 04 D2`.



Als nächstes wollen wir noch zwei Variablen vom Typ `char` anlegen, die jeweils 1 Byte groß sind.



Wir überlegen nun, was uns der &-Operator in C angewendet auf unsere Variablen zurückgibt. Wir wissen schon, dass wir eine Adresse erhalten, aber welche? Das ist ganz einfach. Wir erhalten immer die Adresse vom ersten Byte des Speicherabschnittes, an dem unsere Variable liegt.



Nun kommen die Pointer ins Spiel. Ein Pointer ist eine Variable, die eine Adresse speichern kann. Um mit einem Pointer arbeiten zu können, muss euer Compiler noch wissen, als was er den Speicherinhalt an der Adresse interpretieren soll. Deshalb hat ein Pointer auch einen Typ – genau wie die schon bekannten Variablen auch. Hier die allgemeine Form zum Deklarieren eines Pointers.

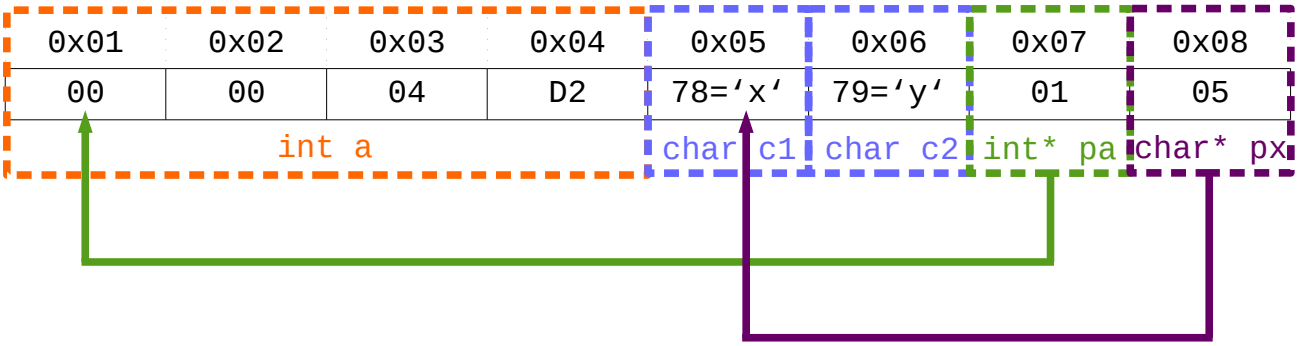
```
type *identifier;   /* 1. Möglichkeit oder */
type * identifier;  /* 2. Möglichkeit oder */
type* identifier;   /* 3. Möglichkeit. Alles äquivalent */

/* Beispiele */

int *pa;  /* ein Pointer namens „pa“, der auf einen int zeigt */
char* s;  /* ein Pointer namens „s“, der auf einen char zeigt */
```

Wir wollen nun zwei Pointer anlegen, die auf Adressen unserer Variablen zeigen.

```
int* pa = &a;   /* pa = 0x01 */
int* px = &c1;  /* px = 0x05 */
```



Wir können nun den Speicher, auf den die Pointer zeigen, interpretieren. Dies nennt man Dereferenzieren eines Pointers. Dafür gibt es den *-Operator. Hier die Schreibweise:

```
*pa;   /* Gehe zur Adresse 0x01 und interpretiere die folgenden 4 Byte
          als int */
*px;   /* Gehe zur Adresse 0x05 und interpretiere das folgende Byte als
          char */
/* Wichtig: (*pa != pa) */
 pa == 0x01;
*pa == 1234;
```

Aber was bringen denn jetzt Pointer? Ganz einfach – wir können den Speicher an einer bekannten Adresse auslesen und manipulieren. Gegeben sei folgender Code-Ausschnitt:

```
int main() {
  int x, y, z;
  f(…);
  return 0;
}
```

Wir wollen die Funktion f benutzen um den Wert von x auf 1, y auf 2 und z auf 3 zu setzen. Ohne Pointer war dies nicht möglich, da f ja nur einen einzigen Wert zurückgeben kann, keinesfalls 3 Werte. Wir übergeben f also die Adressen, an denen unsere Variablen x, y und z liegen, und lassen f den Inhalt des Speichers überschreiben. Hier eine mögliche Implementierung.

```
void f(int* a, int* b, int* c) {
  *a = 1; /* gehe zur Adresse in a und überschreibe den Speicher durch
             einen int mit Wert 1 */
  *b = 2; /* gehe zur Adresse in b und überschreibe den Speicher durch
             einen int mit Wert 2 */
  *c = 3; /* gehe zur Adresse in b und überschreibe den Speicher durch
             einen int mit Wert 3 */
}
```

Unsere main-Funktion passen wir entsprechend an.

```
int main() {
  int x = 0, y = 0, z = 0;
  printf("Vorher: %i %i %i \n", x, y, z);
  f(&x, &y, &z); /* Übergebe die Adressen von x, y und z an f */
  printf("Nachher: %i %i %i \n", x, y, z);
  return 0;
}
```

Die Ausgabe im Terminal sieht dann so aus:

```
Vorher:  0 0 0
Nachher: 1 2 3
```

Wir haben also erfolgreich auf Variablen aus einem anderen Scope heraus zugegriffen. Dieses Prinzip nennt sich „Call-By-Reference“. Dies unterscheidet sich von „Call-By-Value“, welches wir vor der Einführung von Pointern verwendeten. Call-By-Value bedeutet, dass wir, wenn wir eine Funktion mit Argumenten aufrufen, der Funktion Kopien der übergebenen Variablen / Werte übergeben. Call-By-Reference bedeutet, dass wir der Funktion eine Kopie der Adresse einer Variablen übergeben, statt einer Kopie des Wertes einer Variablen. Mit Hilfe der Adresse können wir dann die originale Variable beeinflussen.

```
void f1(int a) {
  a += 4;
}

int main() {
  int x = 1;
  printf("Vorher: %i \n", x);
  f1(x);
  printf("Nachher: %i \n", x);
  return 0;
}
```

Was gibt uns das Terminal in diesem Fall aus?

```
Vorher:  1
Nachher: 1
```
Die Funktion f1 arbeitet nach dem Prinzip Call-By-Value, also wird beim Aufruf von f1 ein Integer übergeben, der eine Kopie des Inhaltes von x hat. Und da f1 nur mit dieser Kopie arbeitet, wird x niemals beeinflusst.

Nun eine kurze Hilfe zur Order of Operations bei Pointern. Ich beziehe mich in der Speicherbelegung auf das obige Beispiel.

```
px++    /* 1) Auswertung von px zu 0x05,
           2) Inkrement von px auf 0x06 */

*(px)++ /* 1) Auswertung von px zu 0x05,
           2) Dereferenzierung von px zu ‘x‘
           3) Änderung des Speichers an 0x05 von ‘x‘ zu ‘y‘ */

*(px++) /* 1) Dereferenzierung von px zu ‘x‘ welches in 0x05 steht
           2) Inkrement von px zu 0x06 */

++px    /* 1) Inkrement von px zu 0x06
           2) Auswertung von px zu 0x06 */

++*(px) /* 1) Inkrement von Wert im Speicher an 0x05
           2) Auswertung zu ‘y‘, welches in Zelle 0x05 steht */

*(++px) /* 1) Inkrement von von px zu 0x06
           2) Dereferenzierung von 0x06 zu ‘y‘ */
```
Offsets bei Pointern sind immer relativ zur Größe des Typs, auf den der Pointer zeigt.

```
pa + 3 == 0x01 + 3 * sizeof(int) == 0x0D
px + 3 == 0x05 + 3 * sizeof(char) == 0x08
```
Zuletzt möchte ich noch den Zusammenhang zwischen Pointer und Array erklären. Angenommen wir haben ein Array aus int-Variablen.

```
int x[4] = {1, 3, 3, 7};
```
Wir haben bisher auf Elemente des Arrays mit dem [ ]-Operator zugegriffen und diesem einen Index übergeben. Was wir bisher nicht wussten: Die Variable x ist ein Pointer auf das erste Element des Arrays im Speicher und der [ ]-Operator ist ein Shortcut für die Dereferenzierung mit einem Offset. Hier die Definition vom [ ]-Operator:

```
int * x;
x[n] == *(x + n);
```
x[n] bedeutet also: Betrachte die Adresse, die in x gespeichert ist. Gehe im Speicher n * sizeof(int) weiter. Interpretiere ab hier sizeof(int) Byte als einen Integer.
Das ist auch ein Grund dafür, warum man Arrays nicht als Value an Funktionen übergeben kann. Die Variable auf ein Array ist nur ein Pointer auf reservierten Speicher. Demnach ist nur Call-By-Reference ohne Umwege möglich.