Pekare (Pointers)

- Variablers innehåll lagras i datorns minne
- Datorns minne är organiserat i form av minnesceller, och varje minnescell har en adress
- En *pekare* “pekar på” en viss adress i datorns minne; via denna pekare kan man komma åt innehållet i denna minnescell
- Pekare används direkt eller indirekt i de flesta programmeringsspråk men i C och C++ har pekare en speciellt framträdande roll
- En förståelse för hur pekare och minneshantering fungerar är nödvändig för att kunna skriva effektiva program (oberoende av vilket språk som används)

Användningsområden för pekare i C

- Tillåt en funktion att *modifiera* variablers innehåll i den anropande funktionen
- Manipulera och traversera (=gå igenom) räckor på ett effektivt sätt
- Dynamisk minneshantering (skapa datastrukturer som växer och krymper vid behov)
- Skicka *funktioner* som parametrar till andra funktioner
- I denna kurs bekantar vi oss med de tre första användningsområdena

Faror med pekare

- Pekare medför stora möjligheter, men också risker
- Buggar pga felaktig användning av pekare ger upphov till fel som kan uppstå och uppträda på ett slumpmässigt sätt
- En felaktigt använd pekare kan peka vart som helst i datorns minne – och den platsen kan variera för varje gång programmet körs
- Beroende på vart pekaren råkar peka vid en viss programkörning kan helt olika fel uppstå

=> svårt att hitta och korrigera felen

Deklaration av pekare

- Det vanligaste sättet att använda pekare är genom att deklarera en *pekare till en variabel*.
- För att deklarera en pekare används en *asterisk* (*****):

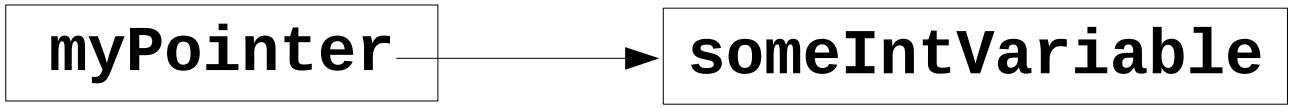
```
int* myPointer; //kan även skrivas int *myPointer
```

- Du har nu en namngiven pekare **`myPointer`** som är avsedd att "peka på" en **`int`**.
- Samma regel som för vanliga variabler: en pekare som inte är initialiserad kan peka vart som helst.

Initialisering av pekare

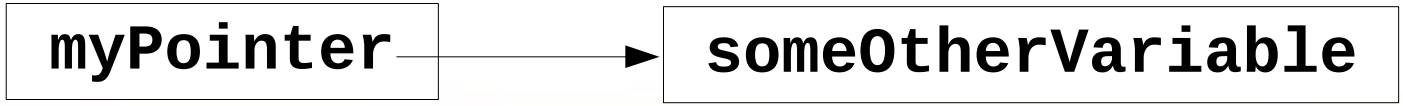
- En pekare initialiseras genom att tilldela den en minnesadress. En variabels minnesadress kan erhållas med adressoperatorn **&**

```
int someIntVariable = 13;
int* myPointer = &someIntVariable;
```



- Pekarens värde kan därefter ändras så att den istället pekar mot en annan variabel:

```
int someOtherVariable = 26;
myPointer = &someOtherVariable;
```



“Dereferencing” av pekare

- Med hjälp av “dereference” eller “indirection”-operatorn ***** kommer man åt *värdet* som finns i den minnesadress som pekaren hänvisar till:

```
int someIntVariable = 13;
int* myPointer = &someIntVariable;
printf("Content of myPointer = %d\n",
                                    *myPointer);
```

- **Viktigt:** Pekaren och variabeln den pekar på är sammanlänkade!

  => Om värdet på variabeln förändras kommer samma förändring att synas även via dereferering av pekaren

  => Värdet på variabeln kan ändras *via pekaren*

“Dereferencing” av pekare

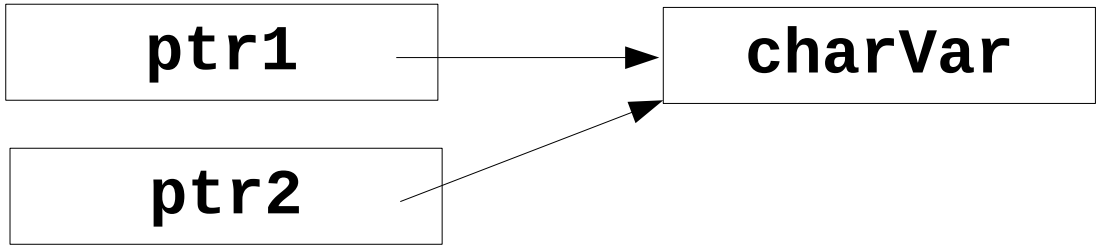
```
int varA = 40;
int varB = varA;

varA = 100;

printf("Value of varA: %d\n",
                          varA);

printf("Value of varB: %d\n",
                          varB);
```

```
int varA = 40;
int* pointer = &varA;

varA = 100;
printf("Value of varA: %d\n",
                            varA);

printf("Value of *pointer: %d\n",
                           *pointer);

*pointer = 80;
printf("Value of varA: %d\n",
                            varA);
```

Två pekare, en variabel

- Två pekare kan peka *till samma variabel:*

```
char charVar = 'a';
char* ptr1 = &charVar;
char* ptr2 = &charVar;
```



Fråga: Vad skulle hända om vi skulle skriva
a) `ptr2 = ptr1` b) `ptr2 = &ptr1` ?

Fråga: Varför är det viktigt att alltid ange *vilken typ av data* en pekare skall hänvisa till? Vore det inte enklare att endast ha en sorts pekare?

NULL-pekare

- En oinitialiserad pekare kan peka vart som helst => Mycket viktigt att initialisera pekare
- Kan ge en pekare startvärdet NULL eller 0
- En funktion som returnerar en pekare returnerar typiskt en 'nullpointer' om operationen misslyckas

NULL-pekare

```
int* pointerUsingFunction()
{
    int* ptr = NULL;
    // ...update ptr if possible
    return ptr;
}

int main(void)
{
    int* result = pointerUsingFunction();
    if (result != NULL ) // Could also write if (result)
    {
        // Use result
    }
    return 0;
}
```

Pekare som funktionsparametrar

- Vad blir resultatet från nedanstående program?

```
void test(int* intPointer) {
    *intPointer = 100;
}

int main(void) {
    int i = 50;
    int* p = &i;
    test(p);
    printf("i is now %d\n", i);
    printf("Dereferencing p gives %d\n", *p);

    return 0;
}
```

Pekare som funktionsparametrar

- Med hjälp av pekare kan en funktion ändra på de parametrar som sänts till den, och dessa ändringar kommer att synas i den anropande funktionen
- Möjliggör att en funktion returnerar mer än ett värde, utan användning av strukturer eller räckor
- Att skicka pekare kan vara mera effektivt än att skicka “normala” variabler, även om dessa inte behöver modifieras
  - Då en struktur skickas som en vanlig parameter måste alla dess medlemsvariabler kopieras – om samma struktur skickas som en pekare behövs inte detta.

Pekare som funktionsparametrar

- Det är inte nödvändigt att skapa en pekarvariabel för att skicka ett värde som en pekare (jfr. **`scanf()`** !)

```
void test(int* intPointer, float* floatPointer) {
    *intPointer = 100;
    *floatPointer = 3.14;
}

int main(void) {
    int intVar = 50;
    float floatVar = 9.2;
    test(&intVar, &floatVar); // Send the address of
                              // the variables
    printf("intVar is now %d\n", intVar);
    printf("floatVar is now %f\n", floatVar);
    return 0;
}
```

Pekare och struct

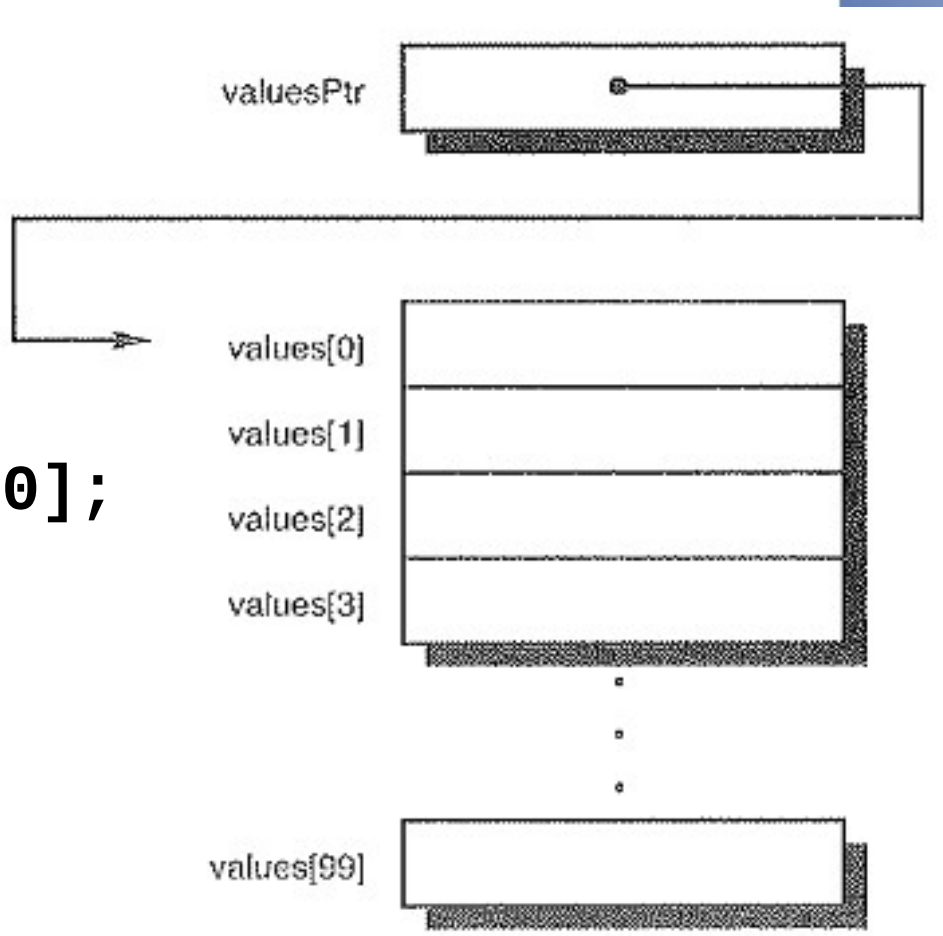
- Kan skicka en pekare till en **`struct`** som parameter
- Två möjligheter för att använda medlemsvariablerna:
  - Dereferencing
  - “Pilsyntax”

```
void test(struct date* dt) {
    (*dt).day=10;   // Dereference
    dt->day=10;     // Arrow-syntax
}

int main(void) {
    struct date christmas = {24,12,2017};
    test( &christmas );
    return 0;
}
```

Pekare och räckor

- I C/C++ är pekare och räckor tätt sammankopplade
- För att skapa en pekare till (början av) en räcka behövs ingen **&**-operator:

```
int values[100];
int* valuesPtr = values;
```



- Kunde också ha skrivit:

```
int* valuesPtr = &values[0];
```

Pekare och räckor

- För att “hoppa” till nästa element i räckan kan man helt enkelt addera en etta till pekaren:

```
int values[100];
int* valuesPtr = values;

valuesPtr++; // Nu pekas det på det 2:a
             // elementet i räckan

valuesPtr = valuesPtr + 10; // ...och nu på
                            // det 12:e

*valuesPtr = 437234; // Det 12:e elementet har
                     // nu värdet 437234
```

Pekare och räckor

- När en räcka skickas som parameter till en funktion är det i verkligheten endast en *pekare* till räckan som skickas.
  - Förklarar varför **`scanf`** inte kräver något **&**-tecken för att läsa in en sträng
  - Förklarar varför innehållet i en räcka kan modifieras från en funktion
- Kan vara mera effektivt att använda sig av pekare istället för index för att gå igenom (traversera) räckor

Is it an array or is it a pointer?

```
int arraySum(int array[], int n) {
    int sum = 0;
    int i = 0;
    for (; i < n; i++) {
        sum = sum + array[i];
    }
    return sum;
}

int main(void) {
    int values[10] = {3, 4, 6, 6, 3, 8, -1, 5, 3, 7};
    printf("The sum is %d\n", arraySum(values, 10));
    return 0;
}
```

Is it an array or is it a pointer?

```
int arraySum(int* array, int n) {
    int sum = 0;
    int* arrayEnd = array+n-1  // Create a pointer to the
                               // last element in the array

    for (; array <= arrayEnd; array++) {
        sum = sum + *array;
    }
    return sum;
}

int main(void) {
    int values[10] = {3, 4, 6, 6, 3, 8, -1, 5, 3, 7};
    printf("The sum is %d\n", arraySum(values, 10));
    return 0;
}
```

Textsträngar och pekare

- Finns vissa skillnader mellan användning av strängpekare respektive räckor:

```
char text[80] = "This is OK!";
text = "Will this work?"; // Not valid, only works when
                          // initializing the string array

text[0] = 'X'; // Valid, can modify the string

char* textPtr = "This is also OK!";
textPtr = "This works fine!"; // Valid
textPtr[0] = 'X'; // Not valid. Will compile but
                  // may cause a run-time crash
                  // Must use dynamic memory
                  // allocation for this to
                  // work!
```

Textsträngar och pekare

```
void copyString( char* from, char* to ) {
    while ( *from != '\0' ) {
        *to = *from;
        to++;
        from++;
    }
    *to = '\0'; // Why is this needed?
}

int main(void) {
    char string1[] = "Copy me!";
    char string2[50];
    copyString(string1, string2);
    printf("string1: %s string2: %s\n", string1, string2);
    return 0;
}
```

Funktioner som returnerar pekare

- Fungerar nedanstående program korrekt?

  (Tips: Lokala variablers giltighetsområden)

```
int* getMagicNumber() {
    int blackMagic = 42;
    return &blackMagic;
}

int main(void) {
    int* result = getMagicNumber();
    printf("Got the magic number: %d", *result);
    return 0;
}
```

Minnesadresser

- Vi kan också skriva ut den minnesadress som pekaren hänvisar till:

```
int a = 39;
int* a_ptr = &a;
printf("Address is %p\n", a_ptr); // Output in
                                  // hexadecimal format
int arr[5] = {1, 2, 3, 4, 5};
int* ptr_to_elem_3 = &arr[2];
int* arr_ptr = arr;
arr_ptr += 2;
printf("%p %p\n", ptr_to_elem_3, arr_ptr); // Output?
```