SP:PROG3 HT10
Programmering i C och C++, 7,5hp

Föreläsning 13
Funktionspekare
Medlemspekare

Litet exempel på funktionspekare

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

void insult(string who){
   cout << who << ", du är så dum och ful!\n";
}

void praise(string who){
   cout << who << ", du är så snygg och begåvad!\n";
}

int main(){
   void (*fpek)(string) = insult;

   fpek("Jozef");
   fpek = praise;
   fpek("Jozef");

   return 0;
}
```

F13: 1

Om funktionspekare

Funktionspekare kan som alla andra pekarvärden lagras i variabler, i datastrukturer (t.ex. en array av funktionspekare), skickas som argument till andra funktioner och returneras från funktioner, m.a.o. användas på alla sätt som andra typer av värden kan användas på.

Funktionspekare gjorde att man redan i C kunde skriva programkomponenter vilkas beteende kunde modifieras och anpassas under exekvering genom att de vid olika tillfällen kunde anropa olika funktioner från samma instruktioner (det som i objektorienterade språk kallas dynamisk bindning och polymorfism).
Dynamisk bindning (överskuggning) är implementerad mha funktionspekare.

Avreferering av en funktionspekare med anropsparenteser efter innebär anrop av den funktion som pekaren pekar ut för tillfället:

```
(*fpek)(3);
```

ANSI-C införde möjligheten att utelämna den explicita avrefereringen:

```
fpek(3);
```

Ett funktionsnamn utan anropsparenteser är adressen till funktionen:

```
fpek=funk;
```

F13: 2

Deklarationssyntax - exempel

Eftersom kompilatorn skall kunna kontrollera argument- och returvärden vid anrop av funktioner även via pekare, måste funktionspekare deklareras med information om dessa. Deklarationssyntaxen följer anropssyntaxen och måste ta hänsyn till operatorernas precedens:

`void (*fpek)(int);` - `fpek` är en pekare till en funktion som tar en `int` som argument och returnerar `void`. Parenteser kring `*fpek` är nödvändiga, `void *fpek(int)` betyder funktion som returnerar `void *`

`char *(*fpek)(char *, int);` - `fpek` är en pekare till en funktion som tar en `char *` och en `int` som argument och returnerar en `char *`

`int (*fpekvek[10])(int);` - `fpekvek` är en vektor med 10 pekare till funktioner som tar en `int` som argument och returnerar en `int`
Anrop av en funktion via pekare i vektorn skulle ske enligt följande exempel:

```
i=fpekvek[3](137);
```

F13: 3

Deklarationssyntax - exempel (forts.)

`int (*getfunk(int))(char *);` - `getfunk` är en funktion som tar en `int` som argument och som returnerar en pekare till en funktion som tar en `char *` som argument och returnerar en `int`

`void (*signal(int sig, void (*hndl)(int)))(int);`
- `signal` är en funktion som tar som argument dels en `int` (sig), dels en pekare (hndl) till en funktion som tar en `int` som argument och returnerar `void`. `signal` returnerar en pekare till en funktion som tar en `int` som argument och returnerar `void`

Sådana deklarationer kan underlättas genom att ett namn för funktionspekartypen definieras med en `typedef`:

```
typedef void (*Sighandl)(int);

Sighandl signal(int sig, Sighandl hndl);
```

När pekar- eller argumentnamn är oväsentliga, t.ex. vid typomvandlingar, kan de utelämnas:

```
(int (*)(int, int))
```

F13: 4

Ännu ett litet exempel på funktionspekare

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

void insult(string who){
      cout << who << ", du är så dum och ful!\n";
}

void praise(string who){
      cout << who << ", du är så snygg och begåvad!\n";
}

void menu(string dummy){
      cout << "0 - quit\n1 - menu\n2 - insult\n3 - praise\n";
}

void quit(string dummy){
      exit(0);
}
```

F13: 5

Ännu ett exempel på funktionspekare, forts.

```
typedef void (*FuncPtr)(string);

FuncPtr funcs[] = {quit, menu, insult, praise};

int main(){
       string name;
       int com;

       cout << "Vad heter du? ";
       cin >> name;
       for(;;){ // "Oändlig" loop, prog avslutas från en funktion
          cout << "Kommando: ";
          cin >> com;
          if (com < 0 || com > 3)
              cout << "Fel kommando!\n";
          else
              funcs[com](name);
       } // for

       return 0;
}
```

F13: 6

Exempel på call-back-funktion

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

bool larger(int x, int y){    // Call-back-funktion att skicka
   return x > y;              // till sorteringen
}

int main(){
   int arr[] = {43, 6, 73, 53, 76, 20};
   const int COUNT = sizeof(arr)/sizeof(int);

       ...
   sort(arr, arr + COUNT);  // Default sortering, stigande

       ...
   sort(arr, arr + COUNT, larger); // Med call-back-funk.
                                   // som jmf-kriterium

   return 0;
}
```

F13: 7

Medlemspekare

Ibland uppstår behovet att peka ut en viss medlem, som skall behandlas i olika objekt. C++ har begreppet medlemspekare (pointer to member) för sådana situationer (har ingen motsvarighet i Java).

En medlemspekare är implementerad som ett offset in i ett objekt till den aktuella medlemmen (hur långt från objektets början medlemmen ligger). För att ge en viss medlem i ett visst objekt måste den kompletteras med ett objekt (namn eller referens) eller en pekare till ett objekt.

Medlemspekare till medlemsfunktioner är mest användbara, men först lite syntax.

Exempel: antag deklarationerna

```
struct Strukt{
   int x, y;
};
Strukt vek[10]={{2,3}, {5, 17}, {3, 8}, ......};
```

Antag att vi vill kunna summera ibland alla `x` i `vek`, ibland alla `y`:

```
int Strukt::*mpek; // Deklaration av medlemspekare till en int i Strukt
if (x ska summeras)
    mpek=&Strukt::x; // "Medlemsadress"-tagning
else
    mpek=&Strukt::y;

int sum=0;
for(int i=0; i<10; i++)
    sum += vek[i].*mpek; // Åtkomst via medlemspekaren
```

F13: 8

Operatorer för medlemspekare

C++ inför tre operatorer för medlemspekare och en notation för ”medlems-adress”-tagning:

`::*`	- deklarator för medlemspekare, t.ex. `int Strukt::*mpek;` Obs alltså att medlemmarnas typ och klassens namn ingår.
”Medlems-adress”-tagning	-”Adressen” till en medlem fås genom notationen `mpek=&Strukt::x;` Obs dock att det är en specialkonstruktion, i själva verket står en medlemspekare för ett offset, inte för en adress. Detta offset måste kombineras med en pekare till ett objekt eller med ett objekt (eller en referens till ett objekt)
`->*`	- åtkomst till en medlem via medlemspekare hos ett objekt som pekas ut av en objektpekare, t.ex. `int i=pek->*mpek;`
`.*`	- åtkomst till en medlem via medlemspekare hos ett objekt (eller en objektreferens), t.ex.: `Strukt obj={5, 7}; int i=obj.*mpeki;`

F13: 9

Pekare till medlemsfunktioner

Exempel:

```
class Valued{
   int value;
public:
   Valued():value(0){}
   void add(int v){value+=v;}
   void sub(int v){value-=v;}
   void mult(int v) {value*=v;}
};

Valued values[10];
void (Valued::*mfunk)(int i); //Pekare till medlemsfunktion

cout << ”Operation värde: ”
char perator; int perand;
cin >> perator >> perand;
switch(perator){
  case '+': mfunk=&Valued::add; break;
  case '-': mfunk=&Valued::sub; break;
  case '*': mfunk=&Valued::mult; break;
}
for(Valued *pek=values; pek<values+10; pek++)
   (pek->*mfunk)(perand);
```

F13: 10

Mer om medlemsfunktionspekare

Medlemsfunktionspekare kan t.ex. läggas i datastrukturer som objekt, `map<>`, arrayer:

```
void (Valued::*mfarr[3])(int) = {&Valued::add, &Valued::sub,
                                 &Valued::mult};
```

Varvid både objektet och operationen kan indexeras:

```
(values[5].*mfarr[1])(3); // Anropar values[5].sub(3);
```

Definitionen av arrayen kan förenklas med `typedef`:

```
typedef void (Valued::*Valfunk)(int);
Valfunk mfarr[3]={&Valued::add, &Valued::sub, &Valued::mult};
```

Oftast görs sådana typnamndefinitionen i klassen varvid klassnamnet syns i arraydefinitionen:

```
class Valued{
public:
   typedef void (Valued::*Mfunk)(int);
   ...
};

Valued::Mfunk mfarr[3]={&Valued::add,&Valued::sub,&Valued::mult};
```

F13: 11