# EL2310 – Scientific Programming

## Lecture 9: Pointers and Structures


Ramviyas Parasuraman (ramviyas@kth.se)

Royal Institute of Technology – KTH

# Overview

## Lecture 9: Pointers and Structures

- Wrap Up
- Function Pointers
- Constant variables
- Structures
- Pointers and Structs
- Memory Allocation

# Wrap up

- Change the scope of variables: `extern` and `static`
- Pointer basics: Pointer contain the address of another variable
  - To pass reference to big things in memory
  - To return multiple values from functions
- Assigning a pointer: `int *b = &a;`
- Deferencing a pointer: `*b = 4;`
- Passing values by reference:

```
void func(double x, double *f);
```

# Wrap up

- Arrays and Pointers:
```
int a[] = {1,2,3,4,5,6,7,8};
int *p = &a[0];
```
- increment/decrement pointers: `p++, p-=3; p[-2] = 2`
- access array values: `*(p+i)` and `p[i]`
- access array address: `&a[i]` and `a+i`
- Pointers to Pointers:
```
int a;
int *p = &a;
int **pp = &p;
```
- `void` and `NULL` pointers

# Pointers

- Comparison
- Let `int *p1, *p2;`
- What is the difference?

```
...
if (p1 == p2) ...
...
if (*p1 == *p2) ...
```

# Announcements

- C project announced. Deadline 30th Sep (Fri).
- Next lab session (help session):
  27th Sep (Tues) 9.30-12, room 304, TK14
- Slight changes in Groups (for presentations). Check updated list in Bilda later today.

# Task 8.2

- Rewrite the Newton code using a function of the following form:
  `void eval_fcn(double x, double *f, double *dfdx);`

# Task 8.3

- Write the function
  `void strcpy2(char *dest, char *src);`
- Should copy the string `src` into `dest`

# Lecture 9: Pointers and Structures

# Pointer to functions

- Just like in Matlab you can work with pointers to functions
- In C you need to declare explicitly what the argument the function has as input and output
- Ex: Pointer (`fcn`) to a function that returns an `int` and takes a `double` as argument

```
int (*fcn)(double)
```

# Arrays of pointers to functions

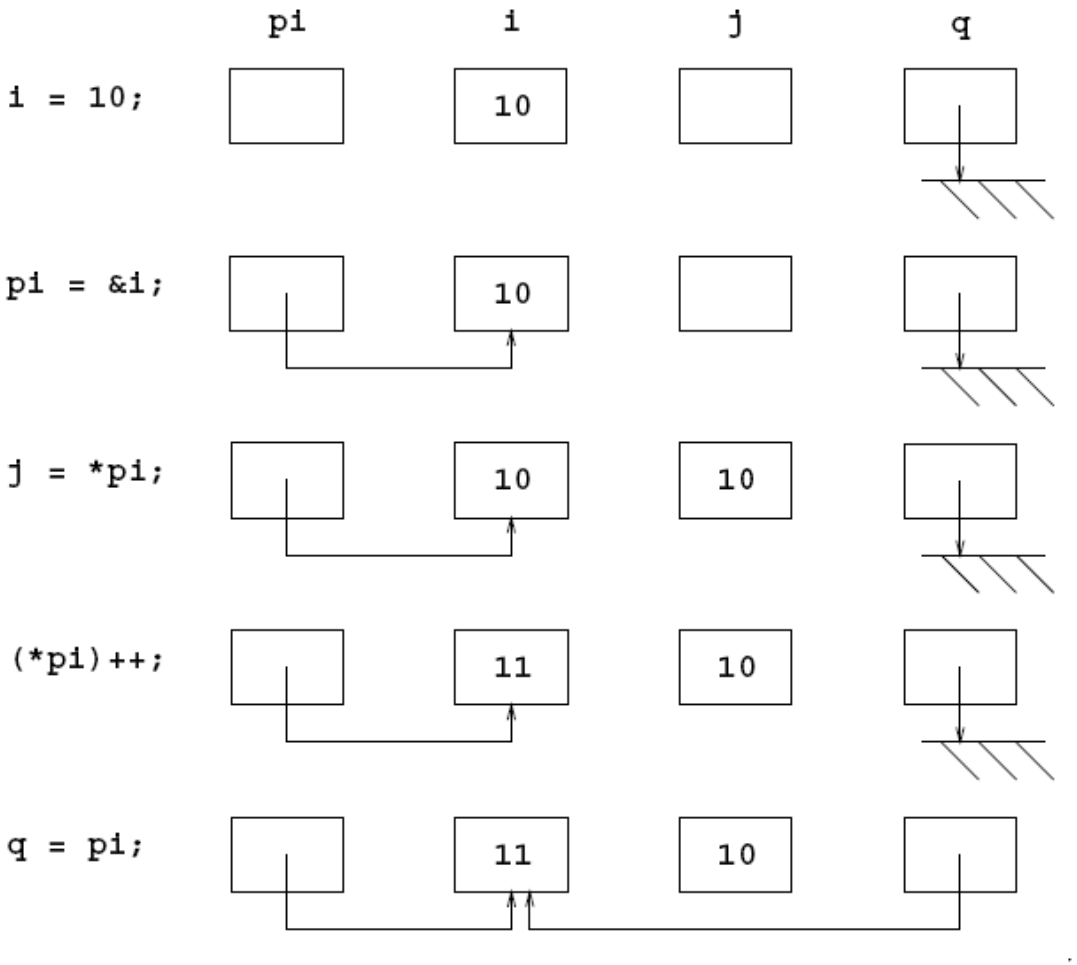
- Can store arrays of function pointers
- To declare an array `pf` of 4 pointers to functions we do
  ```
  double (*pf[4])(double);
  ```
- You assign values by
  ```
  pf[0] = &fcn1;
  ```
- and you use them as
  ```
  pf[0](4.2);
  ```

# Task 1

- Illustrate what happens in the following case

```
int *pi, i, j, *q = NULL;
i = 10;
pi = &i;
j = *pi;
(*pi)++;
q = pi;
```

# Task 1 cont'd



# Task 2

Write a program which accesses the functions,

- `int add(int x,int y){return x+y}`
- `int mul(int x,int y){return x*y}`

using function pointers

# Task 3

- Rewrite the Newton function so that it can take a function pointer instead
- This makes it easier to switch functions

# Task 4

- Write a program with several functions, all with the same interface
- Create an array of pointers to these functions
- Loop through the pointers and call the functions

# Lecture 9: Pointers and Structures

# const

- If you want to make sure that a variable is not changed you can use the `const` keyword
- Ex: `const double pi = 3.1415;`

# Lecture 9: Pointers and Structures

# `struct`

- So far we looked at basic data types and pointers
- It is possible to define your own types
- For this we use a `struct`
- Ex:

```
struct complex_number {
  double real;
  double imag;
};
```

- The variables `real` and `imag` are called *members* of the `struct complex_number`.
- Declaring variables `x,y` of type `complex_number` is done with `struct complex_number x,y;`

# Assigning `struct`

- Can be assign similar to arrays
- `struct complex_number x = { 1.1, 2.4 };`
- Will give the complex number $x = 1.1 + 2.4i$.
- One more example:

```
struct person {
  char *name;
  int age;
};
struct person p1 = {"Jan Kowalski", 38};
```

- Order must be same as in structure, unless:

```
struct person p1 = {.age=38, .name="Jan
Kowalski"};
```

# Accessing members of a `struct`

- If you want to set/get the value of a member you use the "." operator
- Ex:

```
struct complex_number {
  double real;
  double imag;
};
struct complex_number x;
x.real = 1.1;
x.imag = 2.4;
```

# typedef

- `typedef` can be used to give types a new name, like a synonym
- Can introduce shorter names for things
- Ex:

```
struct position {
  double x;
  double y;
};
typedef struct position pos;
```

- Now you can use `pos` instead of `struct position`

# Lecture 9: Pointers and Structures

# Pointers and structures

- You can use pointers to structures
- Ex:

```
struct complex_number x;
struct complex_number *xptr = &x;
```

- To access a member using a pointer we use the "$->$" operator
- Ex: `xptr->real = 2;`
- This is the same as `(*xptr).real` or `x.real = 2;`

# Structures of structures

- You can have any number of levels of structures of structures
- Ex:

```
struct position {
  double x;
  double y;
};
struct line {
  struct position start;
  struct position end;
};
```

# Structures of structures

- Continued.
```
struct line l;
l.start.x = 4;   l.start.y = 6;
l.end.x = 2;     l.end.y = -1;
struct line *lp = &l;
lp->start.y = 42;
```

# Pointers to structures in structures

- Normally you need to declare a type before you use it.
- You can have a pointer to the structure you define
- Ex:
```
struct person {
  char name[32];
  struct person *parent;
};
```

# cast

- Some conversions between types are implicit
- Ex: `double x = 4;`
- In other cases you need to tell the compiler to do this
- Ex: `double fraction = 3 / 4;` will give 0
- Ex: `double fraction = (double)3 / 4;`
- We *casted* 3 from an `int` to a `double`
- **Be careful when casting!**

# Casting pointers

- We can convert also between pointer types
- These are typically allowed with gcc (implicit convertions):

```
int a;
char *pa = &a;
int *b;
char *pb = b;
```

- Will generate a warning, use an explicit cast:

```
int a;
char *pa = (char*) &a;
int *b;
char *pb = (char*) b;
```

- **Be even more careful when casting pointers!**

# Lecture 9: Pointers and Structures

# Dynamic allocation of memory

- Sometimes you do not know the size of arrays when you write code
- Idea: Allocate memory dynamically
- This way you can allocate memory at runtime
- You can calculate how much memory you need and allocate (e.g. array) only then

# malloc

- Allocate memory with `malloc`
- Need to `#include <stdlib.h>`
- This function returns a pointer of type `void*`
- Ex: `int *p = malloc(100*sizeof(int));`
- Will allocate memory for 100 `int`s
- You can use an explicit cast:
  `int *p = (int*)malloc(100*sizeof(int));`

# free

- You should free the memory that you no longer need!!!
- Ex:

```
int *p = (int *)malloc(100*sizeof(int));

...

free(p);
```

- If you do not free allocated memory you will get memory leaks
- Your program will crash eventually
- A big problem if you program should run a very long time

# Common mistakes

- Forgetting to free memory (memory leak!!!)
- Using memory that you have not initialized
- Using memory that you do not own
- Using more memory than you allocated
- Returning pointer to local variable (thus no longer existing)