Pointers to Functions

http://www.newty.de/fpt/fpt.html

Excellent reference/resource
What is a function pointer?

- A pointer, i.e. a variable, which points to the address of a function.

You must keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus a function in the program code is nothing other than an address. It is only important how you, or better your compiler/processor, interprets the memory a pointer points to.
Why use function pointers?

- Function Pointers provide some extremely interesting, efficient and elegant programming techniques.
- You can use them to:
  - replace switch/if-statements,
  - realize your own late-binding (simple discussion later)
  - implement callbacks
- Which leads to:
  - Greater flexibility and better code reuse

Why *not* use function pointers
- Complicated syntax
- Do you really need a function pointer

Why *use* function pointers
- They are less error prone than normal pointers because you will never allocate or deallocate memory with them.
- Run-time options
A pointer variable can be declared as pointing to a function. The declaration of such a pointer is done by,

```c
int (*func_pointer)();
```

The parentheses around `*func_pointer` are necessary, else the compiler will treat the declaration as a declaration of a function.

To assign the address of a function to the pointer, the statement,

```c
func_pointer = lookup; // initialize (use &?)
```

where `lookup` is the function name, is sufficient.

In the case where no arguments are passed to `lookup`, the call is

```c
(*func_pointer)(); // call/use
```

The parentheses are needed to avoid an error.

If the function `lookup` returned a value, the function call then becomes,

```c
i = (*func_pointer)();
```

If the function accepted arguments, the call then becomes,

```c
i = (*func_pointer)( argument1, argument2, argumentn);
```
Let's start with a basic function to point to:

```c
int addInt(int n, int m) {
    return n+m;
}
```

Next, define/declare a pointer to a function which receives 2 ints and returns an int... parentheses needed:

```c
int (*functionPtr)(int,int);
```

Now, point to the function:

- give the pointer an initial value (always):
  ```c
  functionPtr = &addInt;
  ```
- can also be written (and often is) as
  ```c
  functionPtr = addInt;
  ```
- which is also valid since the standard says that a function name in this context is converted to the address of the function (similar to array name being a pointer)

To use a pointer to the function:

```c
int sum = (*functionPtr)(2, 3); // explicitly dereferencing
OR
int sum = functionPtr (2,3);    //using name of function pointer
```
Function pointers in return values

```c
int (*functionFactory(int n))(int, int) {
    printf("Got parameter %d", n);
    int (*functionPtr)(int, int) = &addInt;
    return functionPtr;
}
```

What does this do?

- // this is a function called functionFactory which receives parameter n
- // and returns a pointer to another function which receives two ints
- // and it returns another int
#include <stdio.h>
void my_int_func(int x)
    { printf( "%d\n", x ); }
    // no return stmtt because no return type
int main() {
    void (*foo)(int);
    /* the ampersand is actually optional */
    foo = &my_int_func;
    int a = 1;
    (*foo)(a);
    return 0;
}
int DoIt(float a, char b, char c) {
    printf("DoIt\n"); return a+b+c; }
int DoMore(float a, char b, char c) {
    printf("DoMore\n"); return a-b+c; }

int (*pt2Function)(float, char, char) = NULL;
pt2Function = DoIt; // short form
    OR
pt2Function = &DoMore; // use address operator

if(pt2Function >0){ // check if initialized
    if(pt2Function == &DoIt)
        printf("Pointer points to DoIt\n"); }
else printf("Pointer not initialized!!\n");

int result1 = pt2Function (12, 'a', 'b'); // C short way
    OR
int result2 = (*pt2Function) (12, 'a', 'b'); // more clear
// The four arithmetic operations ... one of these functions is selected at run time
float Plus (float a, float b) { return a+b; }
float Minus (float a, float b) { return a-b; }
float Multiply(float a, float b) { return a*b; }
float Divide (float a, float b) { return a/b; }
// <opCode> specifies which operation to execute
void Switch(float a, float b, char opCode) {
    float result; // execute operation
    switch(opCode) {
        case '+': result = Plus (a, b); break;
        case '-': result = Minus (a, b); break;
        case '*': result = Multiply (a, b); break;
        case '/': result = Divide (a, b); break;
    }
    printf("Switch: 2+5= is %f", result); // display result 
}
Switch-Statement vs Function pointer (cont)

- What if you want to select one function out of a pool of possible functions? Can you pass a function pointer as an argument?

```c
// <pt2Func> is a function pointer and points to a function which takes two floats
// and returns a float. The function pointer “specifies” which operation will execute.

void Switch_With_Function_Pointer(float a, float b, float (*pt2Func)(float, float)) {
    float result = pt2Func(a, b);  // call using function pointer
    printf("Switch replaced by function pointer: 2-5=%f", result);  // display result
}

// Execute example code
void Replace_A_Switch() {
    printf("Executing function Replace_A_Switch\n")
    Switch(2, 5, '+');
    Switch_With_Function_Pointer(2, 5, &Minus); }
```

**Important note:** A function pointer always points to a function with a specific signature! Thus all functions, you want to use with the same function pointer, must have the **same parameters and return-type**!
How to return a function pointer

// Function takes a char and returns a pointer to a function
// which is taking two floats and returns a float.
// <opCode> specifies which function to return

float (*GetPtr1(const char opCode))(float, float) {
    if(opCode == '+') return &Plus;
    else return &Minus; // default if invalid op passed }

// Execute example code

void Return_A_Function_Pointer() {
    // define a function pointer and initialize it to NULL
    float (*pt2Function)(float, float) = NULL;
    pt2Function=GetPtr1('+'); // get function pointer from 'GetPtr1'
    float result1 = (*pt2Function)(2, 4); // call function using pointer
    pt2Function=GetPtr1('-'); // get function pointer from 'GetPtr1'
    float result2 = (*pt2Function)(2, 4); // call function using the pointer
}

result1 = ?        result2 = ?

** NOTE: These two statements are equivalent to: float result1 = GetPtr1(‘+’)(2,4)
Arrays of Function Pointers (ex#1)

Defining and using an array of function pointers offers the option to select a function using an index.

```c
void Array_Of_Function_Pointers() {

    // define arrays and init each element to NULL, <funcArr1> and <funcArr2> are arrays
    // with 10 pointers to functions which return an int and take a float and two char

    // directly defining the array
    int (*funcArr2[10])(float, char, char) = {NULL};

    // assign the function's address - 'DolIt' and 'DoMore' are suitable functions
    // like defined previously
    funcArr1[0] = funcArr2[1] = &DolIt;
    funcArr1[1] = funcArr2[0] = &DoMore; /* more assignments */

    // calling a function using an index to address the function pointer
    printf("%d\n", funcArr1[1](12, 'a', 'b')); // short form
    printf("%d\n", (*funcArr1[0])(12, 'a', 'b')); // "correct" way of calling
    printf("%d\n", (*funcArr2[1])(56, 'a', 'b'));
    printf("%d\n", (*funcArr2[0])(34, 'a', 'b'));
}
```
Arrays of Function Pointers (ex#2)

- C treats pointers to functions just like pointers to data; therefore we can have arrays of pointers to functions.
- This offers the possibility to select a function using an index.
- For example:
  - Suppose that we’re writing a program that displays a menu of commands for the user to choose from. We can write functions that implement these commands, then store pointers to the functions in an array.

```c
void (*file_cmd[]) (void) = {
    new_cmd,
    open_cmd,
    close_cmd,
    save_cmd,
    save_as_cmd,
    print_cmd,
    exit_cmd
};
```

If the user selects a command between 0 and 6, then we can subscript the `file_cmd` array to find out which function to call.

`file_cmd[n]();`
Late/Runtime Binding

- Runtime binding—useful when alternative functions maybe used to perform similar tasks on data (eg sorting)

  - Determine sorting function based on type of data at run time
  - Eg: insertion sort for smaller data sets (n < 100)
  - Eg: Quicksort for large data sets (n > 100000)
  - Other sorting algorithms based on type of data set
Sort Example

- In `<stdlib.h>`, we have a sorting function:
  ```c
  void qsort ( void *base , size_t num , size_t size ,
              int (*comp_func) (const void *, const void *))
  ```

- Consists of three parts
  - a comparison that determines the ordering of any pair of objects
  - an exchange that reverses their order
  - A sorting algorithm that makes comparisons and exchange until the objects are in order.
    `<the sorting algorithm is independent of comparison and exchange operator>

- `qsort` will sort an array of elements. This is a wild function that uses a pointer to another function that performs the required comparisons.
Sort Example

- In `<stdlib.h>`, we have a sorting function:
  ```c
  void qsort ( void *base , size_t num , size_t size ,
              int (*comp_func) (const void *, const void *))
  ```

- Some explanation
  - void * base is a pointer to the array to be sorted. This can be a pointer to any data type.
  - size_t num The number of elements.
  - size_t size The element size.
  - int (*comp_func)(const void *, const void *))This is a pointer to a function.
Sort Example

- qsort thus maintains its data type independence by giving the comparison responsibility to the user.
- The compare function must return integer values according to the comparison result:
  - less than zero: if first value is less than the second value
  - zero: if first value is equal to the second value
  - greater than zero: if first value is greater than the second value
- Some quite complicated data structures can be sorted in this manner.
- The generic pointer type void * is used for the pointer arguments, any pointer can be cast to void * and back again without loss of information.
```c
#include <stdlib.h>
int int_sorter( const void *first_arg, const void *second_arg ){
    int first = *(int*)first_arg; // deref (cast)
    int second = *(int*)second_arg;
    if ( first < second ) {
        return -1;
    } else if ( first == second ) {
        return 0;
    } else {
        return 1;  }
}
int main() {
    int array[10];
    int i;
    /* fill array */
    for ( i = 0; i < 10; ++i ) {
        array[ i ] = 10 - i;  
    }
    qsort( array, 10 , sizeof( int ), int_sorter );
    for ( i = 0; i < 10; ++i ) {
        printf( "%d\n" ,array[ i ] );  
    }
    return 0;
}
```

Reminder: void qsort ( void *base , size_t num , size_t size ,
    int (*comp_func) (const void *, const void *))
Structures

- What is a structure?
  - One or more values, called members, with possibly dissimilar types that are stored together.
  - Used to group together different types of variables under the same name.
  - Aggregates a fixed set of labeled objects, possibly of different types, into a single object (like a record)

- What is a structure NOT?
  - Since members are NOT the same type/size, they are not as easy to access as array elements that are the same size.
  - Structure variable names are NOT replaced with a pointer in an expression (like arrays)
  - A structure is NOT an array of its members so can NOT use subscripts.
struct tag   {member_list} variable_list;

struct S {
    int a;
    float b;
} x;

struct {
    int a;
    float b;
} z;

struct S {
    int a;
    float b;
};

struct S y;

struct S;

Declares x to be a structure having two members, a and b. In addition, the structure tag S is created for use in future declarations.

Omitting the tag field; cannot create any more variables with the same type as z

Omitting the member list declares another structure variable y with the same type as x

Incomplete declaration which informs the compiler that S is a structure tag to be defined later
A struct declaration consists of a list of fields, each of which can have any type. The total storage required for a struct object is the sum of the storage requirements of all the fields, plus any internal padding.

A struct has no place in memory until a variable has been assigned to it.
This declaration introduces the type `struct fraction` (both words are required) as a new type.

- C uses the period (.) to access the fields in a record.
- You can copy two records of the same type using a single assignment statement, however `==` does not work on structs (see note link).

```c
struct fraction {
    int numerator;
    int denominator;
    // can't initialize
};

struct fraction f1, f2;  // declare two fractions
f1.numerator = 25;
f1.denominator = 10;
f2 = f1;  // this copies over the whole struct
```
So tag, member_list and variable_list are all optional, but cannot all be omitted; at least two must appear for a complete declaration.

```
struct {
    int a;
    char b;
    float c;
} x;

struct {
    int a;
    char b;
    float c;
} y[20], *z;
```

Single variable x contains 3 members

- Treated different by the compiler
- DIFFERENT TYPES
- i.e. z = &x is ILLEGAL

So all structures of a given type must be created in a single declaration? NO.

An array of 20 structures (y); and
A pointer to a structure of this type (z)
More Structure Declarations

- The TAG field
  - Allows a name to be given to the member list so that it can be referenced in subsequent declarations
  - Allows many declarations to use the same member list and thus create structures of the same type

```c
struct SIMPLE {
    int a;
    char b;
    float c;
};
```

So: `struct SIMPLE x;`  
`struct SIMPLE y[20], *z;`

Now x, y, and z are all the same kind of structure

Associates tag with member list; does not create any variables
In-class Assignment

How are structure members different from array elements? Consider the type, the name and any memory accessing issues.

Complete the following declaration to initialize x so that the member a is three, b is the string “hello” and c is zero:

```c
struct { int a; char b[10]; float c; } x =
```

Given: struct abc {int a; int b; int c;};

How do you access member a?
**Typedefs** ➔ **typedef** <type> <name>;

---

**Ex1:**

```c
#define true 1
#define false 0
typedef int bool;
bool flag = false;
```

**Ex2:**

```c
char *ptr_to_char; // new variable
typedef char* ptr_to_char; // new type
ptr_to_char a; // new variable
```
A typedef statement introduces a shorthand name for a type. The syntax is...

```c
typedef <type> <name>;
```

- shorter to write
- can simplify more complex type definitions

```c
typedef struct {
    int a;
    char b;
    float c;
} Simple;
```

So ➔ Simple x;
      Simple y[20], *z;

Now x, y, and z are all the same TYPE.

Similar to ➔ int x;
               int y[20], *z;
```c
#include <stdio.h>
typedef struct {
    int x;
    int y;
} point;

int main(void)
{
    /* Define a variable p of type point, and initialize all its members inline! */
    point p = {1,2};
    point q;
    q = p; // q.x = 1 and q.y=2
    q.x = 2;

    /* Demonstrate we have a copy and that they are now different. */
    if (p.x != q.x)
        printf("The members are not equal! %d != %d", p.x, q.x);
    return 0; }
```
Function pointers and Typedef

- We can use function pointers in return values as well
  // this function called functionFactory receives parameter n
  // and returns a pointer to another function which receives
  // two integers and it returns another integer
  int (*functionFactory(int n))(int, int)  
    {  
      printf("Got parameter %d", n);  
      int (*functionPtr)(int, int) = &addInt;  
      return functionPtr;  
    }

- But it's much nicer to use a typedef:
  typedef int (*myFuncDef)(int, int);  
  // note that the typedef name is indeed myFuncDef

  myFuncDef functionFactory(int n)  
    {  
      printf("Got parameter %d", n);  
      myFuncDef functionPtr = &addInt;  
      return functionPtr;  
    }
Original function definition is:

```c
float (*GetPtr1(const char opCode))(float, float)
```

Using a typedef, define a pointer to a function which takes two floats and returns a float:

```c
typedef float(*pt2Func)(float, float);
```

Then you can change the function definition to:

```c
pt2Func GetPtr1(const char opCode)
```

Define an Array of Function Pointers:

```c
int (*funcArr2[10])(float, char, char) = {NULL};
```

Using a typedef:

```c
typedef int (*pt2Function)(float, char, char);
```

You can define an array of function pointers:

```c
pt2Function funcArr1[10] = {NULL};
```
#include<stdio.h>

typedef struct
{  char *name;
    int number;
} TELEPHONE;

int main()
{  TELEPHONE index;
    TELEPHONE *ptr_myindex;
    ptr_myindex = &index;
    ptr_myindex->name = "Jane Doe";  // (*ptr_myindex).name
    ptr_myindex->number = 12345;  // (*ptr_myindex).number
    printf("Name: %s\n", ptr_myindex->name);
    printf("Telephone number: %d\n", ptr_myindex->number);
    return 0; }

What is going on here?
Remember: TELEPHONE is a type of structure;

-> is a “struct member through pointer” operator... see operator precedence (top)
`#include<stdio.h>`
`#include<stdlib.h>`

```c
typedef struct rec
{    int i;
    float PI;
    char A; } RECORD;
```

```c
int main()
{    RECORD *ptr_one;
    ptr_one = (RECORD *) malloc (sizeof(RECORD));
    (*ptr_one).i = 10;
    (*ptr_one).PI = 3.14;
    (*ptr_one).A = 'a';
    printf("First value: %d\n", (*ptr_one).i);
    printf("Second value: %f\n", (*ptr_one).PI);
    printf("Third value: %c\n", (*ptr_one).A);
    free(ptr_one);
    return 0; }
```

“`rec`” is not necessary for given/left code, but *is* necessary for below code update

For below, without RECORD, warning: useless storage class specifier in empty declaration

```c
struct rec *ptr_one;
ptr_one = (struct rec *) malloc (sizeof(struct rec));
ptr_one->i = 10;
ptr_one->PI = 3.14;
ptr_one->A = 'a';
printf("First value: %d\n", ptr_one->i);
printf("Second value: %f\n", ptr_one->PI);
printf("Third value: %c\n", ptr_one->A);
free(ptr_one);
return 0; }
```
how set "pb" to be a pointer to member “b” within structure “mystruct“?
offsetof ➔ tells you the offset of a variable within a structure (stddef.h)

struct mystruct {
  int a;
  char* b; } ;  //note: could put st here instead
struct mystruct st;
char* pb = (char*) &st + offsetof(struct mystruct, b);
Structure memory (again)

What does memory look like?

typedef struct {
    int a;
    short b[2];
} Ex2;

typedef struct EX {
    int a;
    char b[3];
    Ex2 c;
    struct EX *d;
} Ex;

Given the following declaration, fill in the above memory locations:

Ex x = { 10, "Hi", { 5, { -1, 25 } }, 0 };
Ex *px = &x;
In-class exercise

Missing values cause the remaining members to get default initialization... whatever that might be!

typedef struct {
    int    a;
    char   b;
    float  c;
} Simple;

struct INIT_EX  {
    int    a;
    short  b[10];
    Simple  c;
} x = {  10,
          { 1, 2, 3, 4, 5 },
          { 25, 'x', 1.9 }
};

struct INIT_EX  y = { 0 , {10,  20, 30, 40, 50,
                             60, 70, 80, 90, 100 },
                     { 1000, 'a', 3.14 }
    };

Name all the variables and their initial values:
    y.a = 0;
    y.b[0] = 10;  y.b[1] = 20;  y.b[2] = 30;  etc
    y.c.a = 1000;
    y.c.b = 'a';
    y.c.c = 3.14;
More on Structure Declarations

**MEMBERS**
- Any kind of variable that can be declared outside a structure may also be used as a structure member.
- Structure members can be scalars, arrays, pointers and even other structures.

**ACCESS using dot operator**
- Two operands
  - Left = name of structure variable
  - Right = name of the desired member
- Result = the designated member

**OPERATOR PRECEDENCE**
- The subscript and dot operators have the same precedence and all associate left to right.
- The dot operator has higher precedence than the indirection

**Pointer2Structure**
- `->` operator
  - Left = *must* be a pointer to a structure
  - Right = member

**Example**
- `(*sp).a == sp->a`
- Indirection built into arrow/infix operator
- Follow the address to the structure

```c
struct COMPLEX {
    float    f;
    int      a[20];
    long     *lp;
    struct   SIMPLE  s;
    struct   SIMPLE  sa[10];
    struct   SIMPLE  *sp;
} cmplx, cmp[10];
```
```c
struct SIMPLE {
    int a;
    char b;
    float c;
};

struct COMPLEX {
    float f;
    int a[20];
    long *lp;
    struct SIMPLE s;
    struct SIMPLE sa[10];
    struct SIMPLE *sp;
} cmplx, cmp[10];

cmplx.a[1] = 1;
cmplx.s.a = 2;
cmplx.sa[1].b = 'A';
cmplx.sp = &cmplx.s;
cmp[1].f = 3.14;
cmp[5].s.a = 3;
cmp[7].sa[2].b = 'B';

int z = cmplx.a[1];
int j = cmplx.s.a;
char k = cmplx.sa[1].b;
int x = cmplx.sp->a;
float r = cmp[1].f;
int t = cmp[5].s.a;
char y = cmp[7].sa[2].b;
```
Self-Referential Structures

Illegal - infinite

```c
struct SELF_REF {
    int a;
    struct SELF_REF b;
    int c;
};
```

Watch out

```c
typedef struct {
    int a;
    struct SELF_REF *b;
    int c;
} SELF_REF;
```

Correction

```c
struct SELF_REF {
    int a;
    struct SELF_REF *b;
    int c;
};
```

Correction

```c
typedef struct SELF_REF_TAG {
    int a;
    struct SELF_REF_TAG *b;
    int c;
} SELF_REF;
```
Incomplete Declarations

- Structures that are mutually dependent
- As with self referential structures, at least one of the structures must refer to the other only through pointers
- So, which one gets declared first???

```c
struct B;

struct A {  
    struct B *partner;  
    /* etc */
} ;

struct B {  
    struct A *partner;  
    /* etc */
} ;
```

- Declares an identifier to be a structure tag
- Use this tag in declarations where the size of the structure is not needed (pointer!)
- Needed in the member list of A

- Doesn’t have to be a pointer
Structures as Function arguments

- Legal to pass a structure to a function similar to any other variable but often inefficient

```c
/* electronic cash register individual transaction receipt */
define PRODUCT_SIZE 20;
typedef struct {
    char product[PRODUCT_SIZE];
    int qty;
    float unit_price;
    float total_amount;
} Transaction;

void print_receipt (Transaction trans) {
    printf("%s\n, trans.product);
    printf(%d @ %.2f total %.2f\n", trans.qty, trans.unit_price, trans.total_amount);
}
```

- Function call:
  - print_receipt(current_trans);
  - Copy by value copies 32 bytes to the stack which can then be discarded later

- Instead...
  - (Transaction *trans)
  - trans->qty
  - trans->unit_price
  - trans->total_amount
  - print_receipt(&current_trans);
  - void print_receipt(Transaction *trans)