Generic Programming in C

The Goal

- To write code once that works on a variety of types.
- The tools:
  - pointers to functions
  - void*
  - polymorphism (C++, not called Generic programming)
  - templates (C++, “pure” Generic programming)

void*

A way to pass data of an arbitrary type.

Rules regarding void *

void* is a generic pointer capable of representing any pointer type.

```
int i= 5;
double f= 3.14;

void* p;
p= &i; // ok
p= &f; // ok
```

Rules regarding void *

void* pointer cannot be dereferenced.

```
int i= 5;
double f= 3.14;

void* p;
p= &i; // ok
p= &f; // ok

printf("%f\n",*p)); // ERROR
```

Rules regarding void *

void* can be explicitly cast to another pointer type (for example here it's double*)

```
int i= 5;
double f= 3.14;

void* p;
p= &i; // ok
p= &f; // ok
printf("%f\n",*((double*)p)); //   ok
```

void* example: swap_e.c

Pointers to Functions

- Assuming the function f is defined, &f and f are pointers to the function.
- i.e: The address where the function's definition begins in memory.

Example

```
int avg(int num1, int num2) {
    return ( num1 + num2 ) / 2;
}

int (* func)(int, int); // a ptr variable
func = &avg; // assignment
func = avg;   // same
int result = (*func)(20, 30); // invoke it
result = func(20, 30);        // same
```

Suggestion

- Use **typedef** !

```
typedef int (* TwoIntsFunc) (int, int);
TwoIntsFunc f1;
f1 = &avg;
...
f1 = &sum;
```

What is it good for?

- **Generic programming -** pass a function name to another function as a parameter.
- Once upon a time it was a way to make a c struct kind of a poor class.

qsort

```
#include <stdlib.h>
```

Array to be sorted

elements in array

sizeof each element in array

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

Pointer to the comparator function.
Return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

qsort_e.c

qsort problems

- Not efficient – casting, calls to functions.
- Not user freindly.
- Type safety problems.

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

C++ solves all this problems

- efficient – no casting, usually no calls to functions.
- User friendly - sort(arr,arr+ARR_SIZE)
- Type safety.
- Current implementation is a different algorithm – introspective sort.
- We will learn more about it in C++ course