# C Basics 13: Higher-Order Functions

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Multidimensional Arrays

# C Basics 13: Higher-Order Functions

11. Dynamic and Static Memory
12. Structs (Structures)
13. **Higher-Order Functions**
    - **higher-order functions**
    - **function pointer parameters**
    - **library examples**
    - **statement expressions**
14. Variadic Functions
15. Miscellaneous

# Higher-Order Functions

**Higher-order functions** is the name given to functions that *take functions as arguments*.

Higher-order functions are common in mathematics:
e.g., $sum(m, n, f) \equiv \sum_{i=m}^{n} f(i)$

They are a key type of **abstraction mechanism** for algorithms.

We don't have to write: $sum\_i(m, n)$, $sum\_i^2(m, n)$, $sum\_i^3(m, n)$, etc., because $sum(m, n, f)$ covers them all, by abstracting out what is common to all.

Higher-order functions are heavily used in "functional languages" like Lisp and Scheme.

Function parameters that are themselves functions are termed **procedural parameters** (though this term is not used in C).

# Function Pointer Parameters

C allows functions to have *parameters that are functions*.

Technically, what gets passed to the function is a **function pointer**: a pointer to the function definition.

Thus, C *procedural parameters* are often termed **function pointer parameters**.

When the function pointer parameter name is used in a function call, the function pointer is *implicitly dereferenced* to invoke the passed function.

Function pointer parameters provide C with a capability that Java lacked until recently.

Note that *function arguments* are also often referred to as **callbacks**.

# Higher-Order Functions in C

Higher-order functions can be defined in C by using **function pointer parameters**.

Their power is more limited than in a language like Lisp, however:

- C functions are not **first-class objects**
- C does not support **closures**
- C's type checking makes it difficult to use higher-order functions as **generic functions** (functions that can take different argument *types*—i.e., a form of **polymorphism**)
- the parameters of function pointer parameters will often have to be of type `void*`, requiring explicit *dereferencing* and explicit computation of *array offsets*
- function parameter arguments must be specified with constants; one cannot evaluate a variable to retrieve the function name and then get its address.

# Defining Function Parameters

A *function pointer parameter* is specified with notation like:
`rettype (*funcname)(funcparam1, funcparam2,...)`

- `rettype` is the function parameter's *return type*
- `funcname` is the functiona parameter's name, used to call the argument function
- `funcparam1` is the function parameter's first parameter spec

Example: `int (*strtoint)(char *str)`

Specifies a function pointer parameter that has:

- `int` return type for function
- single `char*` parameter for function
- **formal parameter** name `strtoint`

# Example Higher-Order Function: summation

A classic higher-order function most people are familiar with is *summation*, which is often written as: $sum(m, n, f) \equiv \sum_{i=m}^{n} f(i)$

We can implement summation in C as:

```
int summation(int start, int end, int (*func)(int))
{
  int sum = 0;
  for (int i=start; i<=end; i++)
    sum += func(i);  //Note parameter func used in function call

  return sum;
}
```

# Example: summation (contd.)

`summation()` would be used as follows:

```
int intident(int x)
{
  return x;
}

int intsquare(int x)
{
  return x * x;
}


int main()
{
  int sumfirst10 = sum(1,10,intindent);

  int sumfirst5sq = sum(1,5,intsquare);

  ...
}
```

# Standard Higher-Order Functions

Several library and system call functions are higher-order functions, including:

- `int atexit(void (*function)(void))`
- `int on_exit(void (*function)(int, void *), void *arg)`
- ```
  void qsort(void *base, size_t nmemb, size_t size,
             int(*compar)(const void *, const void *))
  ```
- ```
  int pthread_create(pthread_t *thread,
                     const pthread_attr_t *attr,
                     void *(*start_routine)(void *), void *arg)
  ```
- `void (*signal(int signum, void (*handler)(int)) ) (int)`

`signal()`'s definition is particularly confusing because it *returns a function pointer*, so it is typically defined using `typedef`:

```
typedef void (*sighandler_t)(int)
sighandler_t signal(int signum, sighandler_t handler)
```

# Example: pthread_create()

E.g., `pthread_create()` takes a `void*` argument, which will be passed to its `start_routine` function (that has a `void*` parameter).

`void*` parameters are often used with function parameters because they allow any type of data to be passed to the function argument.

This allows different `start_routine()`'s to have very different input requirements and yet still match a fixed prototype.

To pass data to `start_routine()`, an appropriate `struct` is defined and instantiated, and the instantiation's address/pointer passed to `pthread_create()` (*casting* as needed).

# Example: qsort()

`qsort()` is a *higher-order function* that is also a **generic function**:

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

*Generic sorting functions* are another classic application for higher-order functions.

`qsort` allows an array of *any type* of data to be sorted, as long as a **binary comparison function** (basically $\leq$) exists for the type.

The important insight is that the comparison function will be passed *void pointers* to the two objects to be compared, so the argument pointers must be:

1. *cast* to the correct pointer type
2. *dereferenced* to obtain the objects to compare

# Example: qsort() (contd.)

The man page for `qsort()` shows how to define a compatible comparison function for strings, using `strcmp()` to compare the two strings:

```
cmpstringp(const void *p1, const void *p2)
{
  /* The actual arguments to this function are "pointers to
     pointers to char", but strcmp(3) arguments are "pointers
     to char", hence the following cast plus dereference */

  return strcmp(* (char * const *) p1, * (char * const *) p2);
}
```

# Example: qsort (contd.)

A comparison function for `int`'s could be written as:

```
cmpintp(const void *p1, const void *p2)
{
  int x = *(const int *)p1;
  int y = *(const int *)p2;

  if (x == y)
    return 0;
  else if (x < y)
    return -1;
  else
    return 1;
}
```

# Example: qsort() (contd.)

It would be used as follows:

```
int main()
{
  int iarr[] = {5,2,4,5,8,1};

  ...

  qsort(iarr,6,sizeof(int),cmpintp);

  ...
}
```

# Statement Expressions

As an extension, GCC allows **compound statement** to be used as an **expression** if *enclosed in parentheses*.

This allows you to use loops, switches, local variables, etc. within an expression—i.e., to implement a **functional programming** style.

(A *compound statement* is a sequence of statements surrounded by *braces*, while an *expression* has a value (rvalue).)

The *final statement* in the compound statement expression must be an *expression* (followed by a semicolon).

The value of the final expression becomes the value of the entire compound statement expression.

# Statement Expressions (contd.)

Example:

```
static char *strings[NUMSTRINGS];
...
if ( ({int retrn=0;
       for(int i=0;i<NUMSTRINGS;i++)
         if(strings[i]!=NULL){retrn=1; break;}
       retrn;}) )
  printf("strings contains some strings\n");
else
  printf("strings is empty!\n");
```

This is basically the code to check if an array of strings contains any strings or not, written right inside of the `if` clause.

(The expression is written on multiple lines only to make it readable for the slides.)

# Statement Expressions (contd.)

This avoid having to do:

```
static char *strings[NUMSTRINGS];
...
int notempty=0;
for(int i=0;i<NUMSTRINGS;i++)
  if(strings[i]!=NULL) {
    notempty=1;
    break; }
if (notempty)
  printf("strings contains some strings\n");
else
  printf("strings is empty!\n");
```

# Statement Expressions (contd.)

Or having to define a function:

```
int contains(char *strarr[], int arrsize)
{
  for(int i=0;i<arrsize;i++)
    if(strarr[i]!=NULL)
      return 1;
  return 0;
}
```

And then call it just once:

```
if (contains(strings,NUMSTRINGS))
  printf("strings contains some strings\n");
else
  printf("strings is empty!\n");
```