# NWEN 241
# C Functions and Arrays

Qiang Fu

School of Engineering and Computer Science
Victoria University of Wellington

## This Lecture

- Why functions
- How to use functions
- A little bit about pointers

## Functions in C Programs

- Every C program has at least one function: main()
- No C program **needs** to have more than one function in it
  - Everything can be put in main():

## Functions in C Programs

- Every C program has at least one function: main()
- No C program **needs** to have more than one function in it
  - Everything can be put in main(): not a good idea
- Any C program with only a main function is almost certainly for training purposes
- What are functions good for?
  - structuring our thoughts (structured programming)
  - allowing us to re-use code, reducing work and reducing errors
- A C program can be modularised by functions
  - A big program can be broken down into a number of smaller ones

## Creating a Simple Function

- Suppose we frequently wanted to compare two integers and then use the larger. We might have code like this repeatedly written in our program:

```
int p, q, l;
...                 /* p, q initialised */
if (p > q)
  l = p;
else l = q;
...                 /* l gets used */
```

## Creating a Simple Function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: larger()
  - Specify what type of variables that larger() is going to compare:

## Creating a Simple Function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: larger()
  - Specify what type of variables that larger() is going to compare: larger(int, int)
  - Specify what type of value that larger (int, int) is going to return:

## Creating a Simple Function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: larger()
  - Specify what type of variables that larger() is going to compare: larger(int, int)
  - Specify what type of value that larger (int, int) is going to return: int larger(int, int)
  - **int larger(int, int)**: this is called function prototype / declaration

## Creating a Simple Function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: larger()
  - Specify what type of variables that larger() is going to compare: larger(int, int)
  - Specify what type of value that larger (int, int) is going to return: int larger(int, int)
  - **int larger(int, int)**: this is called **function prototype** / declaration
- Make it real: function definition/implementation

## Creating a Simple Function

- Function definition

```
int larger(int x, int y)
{
  if (x > y)
    return x;
  else return y;
}
```

- x and y are called “formal parameters”, whose scope is the body of the function.

## Creating a Simple Function

- Let us use larger()

```
...
int main(void)
{
  ...
  l = larger(p, q); /* p and q are called “actual */
  ...               /* parameters”. Their values are */
}                   /* going to be copied to x and y. */

int larger(int x, int y)
{
  if (x > y)
    return x;
  else return y;
}
```

## Creating a Simple Function

- Let us use larger()

```
...
int main(void)
{
  ...
  l = larger(p, q); /* p and q are called “actual */
  ...               /* parameters”. Their values are */
}                   /* going to be copied to x and y. */

int larger(int x, int y)
{                   /* x and y (NOT p and q) are */
  if (x > y)        /* going to be compared here. */
    return x;       /* the larger value is going to be */
  else return y;    /* returned to larger(p, q). */
}
```

## Creating a Simple Function

- Function prototype

```
int main(void)
{ ...
  l = larger(p, q);     /* larger() not declared yet */
  ...
}
int larger(int x, int y)
{ ...
}
```

  – This is not good …

  – Use function prototype to declare the function before being used

```
int larger(int, int);

int main(void)
{...}

int larger(int x, int y)
{...}
```

## Creating a Simple Function

- Function invocation/call: `l = larger(p, q);`
- Call by value
  - The values of “actual parameters” (p, q) are copied to “formal parameters” (x, y)
  - “actual parameters” and “formal parameters” are separate entities
  - What happens thereafter to “formal parameters” has nothing to do with “actual parameters”
    - Any changes on x, y will not be transferred back to p, q

## Another Simple Function

- Let us swap the values of two variables:

```
...          /* p, q, tmp declared */
...          /* p, q initialised */
tmp = p;
p = q;
q = tmp;
```

- Let us turn this into a function.
  - Tell me the types

## Another Simple Function

- A function for swapping
  - The function does not return a value
  - What is the return type then: **void**

```
void swap(int, int); /*function prototype*/
void swap(int x, int y)
{
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
```

## Another Simple Function

- Does it work?

```
int main(void)
{ int p = 40;
  int q = 80;
  swap(p, q);     /* the values of p, q */
  return 0;       /* are copied to x, y */
}
void swap(int x, int y)
{ int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
```

## Another Simple Function

- Does it work?

```
int main(void)
{ int p = 40;
  int q = 80;
  swap(p, q);     /* the values of p, q */
  return 0;       /* are copied to x, y */
}
void swap(int x, int y)
{ int tmp;
  tmp = x;
  x = y;
  y = tmp;        /* x, y get swapped */
}
```

## Another Simple Function

- Solution: pass in the *addresses* of p, q
  - &p is the address of the memory that stores p’s value
  - The values of p, q are stored at &p, &q
  - We use *pointers* to store the addresses of p, q

```
int *ptrp, *ptrq;   /* declare pointers */
ptrp = &p; /* &p stored in ptrp */
ptrq = &q; /* &q stored in ptrq */
```

  - *ptrp, *ptrq give us access to the values stored at &p, &q

```
printf(“p=%d; q=%d”, *ptrp, *ptrq);
tmp = p;
*ptrp = q;   /* equivalent to p=q; */
*ptrq = tmp; /* p, q get swapped */
```

## Another Simple Function

- The function

```
void swap(int *, int *);
void swap(int *ptrx, int *ptry)
{ int tmp;
  tmp = *ptrx;
  *ptrx = *ptry;
  *ptry = tmp;
}
```

## Another Simple Function

- Let us do the swap

```
int main(void)
{ ...
  int *ptrp, *ptrq;
  ptrp = &p;
  ptrq = &q;
  swap(ptrp, ptrq); /*the addresses of p, q*/
  return 0;         /*are passed to swap() */
}
void swap(int *ptrx, int *ptry)
{ int tmp;
  tmp = *ptrx;
  *ptrx = *ptry;    /*the values stored at */
  *ptry = tmp;      /*the addresses of p, q*/
}                   /*are swapped */
```

## Another Simple Function

- Call by reference
  - The values of “actual parameters” (ptrp, ptrq) are copied to “formal parameters” (ptrx, ptry)
  - The values are memory addresses
  - “actual parameters” and “formal parameters” hold the addresses of the same memory blocks
  - *ptrx, *ptry give you the access to the memory
    - Any changes on *ptrx, *ptry change the values stored in the memory
- We will talk more about pointers

## Functions as Arguments

- A function (*guest function*) can be passed, as an argument, to another function (*host function*)

```
int host_f(int guest_f(int, int), int);
```

## Functions as Arguments

- An example
  - We have made a larger()

```
int larger(int x, int y)
{ if (x > y)
    ...
}
```

  - Let us make a smaller()

```
int smaller(int x, int y)
{ if (x < y)
    return x;
  else
    return y;
}
```

## Functions as Arguments

- Let the larger minus the smaller

```
int l_minus_s(int l(int, int), int s(int,
  int), int x, int y)
{ return(l(x,y)-s(x,y));
}
```

- Invoke the function

```
int main(void)
{ ...
   l_s = l_minus_s(larger, smaller, p,q);
   ...
}
```

## Functions as Arguments

- Did pointers get involved?
  - When a function is used as an argument, gcc interprets it as a pointer

```
int l_minus_s(int l(int, int), int s(int,
  int), int, int); /* l, s are pointers */

l_s = l_minus_s(larger, smaller, p,q);
          /* larger, smaller are pointers */
```

  - int i(int,int) is equivalent to int (*i)(int,int)
    - i is a pointer to a function that takes two int arguments and returns an int
  - We will talk more about pointers later on

## Recursive functions

- A function that calls itself
- A typical example is factorial

```
/*
 * n is a natural number greater than 0
 * n! = n × (n − 1) × (n − 2) ... × 1
 * n! = n × (n − 1)!
 */

int fac(int n)
{ if (n == 0) return 1;
   return n * fac(n-1);
}
```

## Next Lecture

- Arrays and pointers