Functions and pointers

Stat 580: Statistical Computing

- Theme: Black - White
- Printable version

References

- "The C programming language" by Brian W. Kernighan and Dennis M. Ritchie.
- Part of this slide set is based on *Essential C* by Nick Parlante:

(for copyright reason, this notice is reproduced here.)

Functions

Most languages have a construct to separate and package blocks of code.

- C uses "function" to package blocks of code.
- The special function called "main" is where program execution begins.

A C function has

- a name
- a list of arguments which it takes when called
- the block of code it executes when called

We will refer a function by its name followed by `()`. For example, `prod()`.

An example

```c
/* Computes product of two numbers. */
int prod(int i, int j) {
  int k;
  k = i * j;
  return k;
}
```

Elements:

- name: `prod`
- list of arguments: `i`, `j`
- the block of code it executes when called: `{...}`

An example

```c
/* Computes product of two numbers. */
int prod(int i, int j) {
  int k;
  k = i * j;
  return k;
}
```

Syntax:

- `int prod(int i, int j)`
  - the beginning `int` defines the type of the return `k`
  - elements in `(...)` are separated by `,` and each of them takes the form of variable declaration:
    - `int i`: defines an input `i` which is of type `int`
    - `int j`: defines an input `j` which is of type `int`

An example

```c
/* Computes product of two numbers. */
int prod(int i, int j) {
  int k;
  k = i * j;
  return k;
}
```

Syntax:

- `return k;`
  - computes the return value `k` and exits the function
  - execution resumes with the caller
  - there can be multiple return statements within a function

An example

```
/* Computes product of two numbers. */
int min(int i, int j) {
  if (i<j)
    return i;
  else
    return j;
}
```

`void`

If the function does not return anything, you can use `void` as the type of return:

```
void fun(int i);
```

Why would one want a function without any return?

(e.g., use of pointers and global variables, which are covered later.)

Function prototypes

Try:

```
#include<stdio.h>

int main() {
  int i=3;
  sayhello(i);
  return 0;
}

void sayhello(int times) {
  int i;
  for (i=1; i<=times; i++)
    printf("hello!\n");
}
```

You have to declare function before it is called.

Function prototypes

Instead, you can use:

```c
#include<stdio.h>

void sayhello(int times); /* function prototype */

int main() {
  int i=3;
  sayhello(i);
  return 0;
}

void sayhello(int times) {
  int i;
  for (i=1; i<=times; i++)
    printf("hello!\n");
}
```

- a function prototype is a declaration of a function
- in this example, the declaration `void sayhello(int times);` says that `sayhello()` is a function that expects an `int` argument and returns nothing.

Function prototypes

Consider a function without any argument:

```c
#include<stdio.h>

void sayhello() {
  printf("hello!\n");
}

int main() {
  sayhello(123); /* warning */
  return 0;
}
```

vs.

```c
#include<stdio.h>

void sayhello(void) {
  printf("hello!\n");
}

int main() {
  sayhello(123); /* error */
  return 0;
}
```

Function prototypes

We will get error:

```c
#include<stdio.h>

void sayhello(void);

int main() {
  int i=3;
  sayhello(i);
  return 0;
}

void sayhello(void) {
  printf("hello!\n");
}
```

Function prototypes

Warning or error?

```c
#include<stdio.h>

void sayhello();


int main() {
  int i=3;
  sayhello(i);
  return 0;
}

void sayhello(int times) {
  int i;
  for (i=1; i<=times; i++)
    printf("hello!\n");
}
```

Function prototypes

Warning or error? It is legal!

```c
#include<stdio.h>

void sayhello(); /* nothing is assumed about the arguments, all parameter
                    checking is turned off */

int main() {
  int i=3;
  sayhello(i);
  return 0;
}

void sayhello(int times) {
  int i;
  for (i=1; i<=times; i++)
    printf("hello!\n");
}
```

Function prototypes

```c
#include<stdio.h>

void sayhello(); /* nothing is assumed about the arguments, all parameter
                    checking is turned off */

int main() {
  int i=3;
  sayhello(i); /* no checking */
  return 0;
}

void sayhello(void) {
  printf("hello!\n");
}
```

Storage classes

Other than type, C variables have another attribute called storage class.

- storage class indicates how the variable's memory is allocated and the scope of the variable
- the scope of a name is the part of the program within which the name can be used
- four storage classes:
- *auto*: local, within function, created everytime the function is called
- *extern*: global, but surrender to local dominance, created once
- *static*: local, value will be retained, only created once
- *register*: register is where CPU can perform some calculation. For performance purpose.

An example

```c
#include<stdio.h>

double a=1.0, b=2.1; /* extern Storage Class */
int count=0;
/* a = 1.0; */ /* doesn't work */
/* b = 2.1; */ /* doesn't work */

void add1(void) {
  static int count=0;
  double b=0; /* local dominance */
  count += 1;
  printf("Number of times that add1() is called: %d\n", count);
  a = a + 1.0;
  b = b + 1.0;
}

int main(){
  register int i;

  for (i=0; i<5; i++){
    printf("a = %f, b = %f\n", a, b);
    add1();
  }
  printf("count in global: %d\n", count);

  return 0;
}
```

More about external variable

- The scope of an external variable or a function lasts from the point at which it is declared to the end of the file.

```c
#include <stdio.h>

void printx(void);
void addone(void);

int main(){
  addone(); printx();
  /* printf("x=%d\n", x); */ /* undeclared x */
  return 0;
}

int x=0;

/* print x */
void printx(void){
  printf("x=%d\n", x);
}

/* add one to x */
void addone(void){
  x++;
}
```

Extern declaration vs Extern definition

- A declaration gives the properties of a variable (primarily its type)
- A definition causes storage to be set aside additionally
- if `int x;` appears outside of any function, it defines the external variable `x`
  - cause storage to be set aside
- if `extern int x;` appears within a function, it declares for the rest of the function.
- initialization of an external variable goes only with the definition

Extern declaration vs Extern definition

```c
#include <stdio.h>

void printx(void);
void addone(void);

void printx2(void){
  extern int x; /* extern declaration */
  printf("x=%d\n", x);
  x = 0;
}

int main(){
  addone(); printx2(); printx();
  return 0;
}

int x=0; /* extern definition */

/* print x */
void printx(void){
  printf("x=%d\n", x);
}

/* add one to x */
void addone(void){
  x++;
}
```

Block structure

- Variables declared in this way hide any identically named variables in outer blocks, and remain in existence until the matching right brace.

```
#include <stdio.h>

int main(){
  int i=1;
  printf("outer layer: i=%d\n", i);
  if (1){
    int i=2;
    printf("middle layer: i=%d\n", i);
    {
      int i=3;
      printf("inner layer: i=%d\n", i);
    }
    printf("middle layer: i=%d\n", i);
  }
  printf("outer layer: i=%d\n", i);
}
```

Remark on initialization

Without explicit initialization:

- external and static variables are initialized to zero
- automatic and register variables have undefined initial values

```c
#include <stdio.h>

int x;
int main(){
  int y;
  printf("x=%d, y=%d\n", x, y);
  return 0;
}
```

Remark on initialization

- For external and static variables, the initializer must be a constant expression
- For automatic and register variables, it is done every time the function or block is entered. The initializer is not restricted to being a constant.

```c
#include <stdio.h>

int x=1;
/*int y=x+1; */ /* error */
int y=1+1; /* okay */

int main(){
  int z=x+1;
  printf("x=%d, y=%d, z=%d\n", x, y, z);
  return 0;
}
```

Pass by value

```c
int prod(int i, int j) {
  int k;
  k = i * j;
  return k;
}

int main() {
  int a, b, c, k;
  a = 1;
  b = 2;
  k = 0;
  c = prod(a, b); /* Call the function prod with input a and b */
  printf("a=%d, b=%d, c=%d, k=%d\n", a, b, c, k);
  return 0;
}
```

- Some vocabularies:
  - actual parameter: the expression passed to a function by its caller
    - e.g. `a` and `b`
  - formal parameter: the parameter storage local to the function
    - e.g. `i` and `j` in `prod()`

Pass by value

```c
int prod(int i, int j) {
  int k;
  k = i * j;
  return k;
}

int main() {
  int a, b, c, k;
  a = 1;
  b = 2;
  k = 0;
  c = prod(a, b); /* Call the function prod with input a and b */
  printf("a=%d, b=%d, c=%d, k=%d\n", a, b, c, k);
  return 0;
}
```

- assignment operation (=) from each actual parameter to set each formal parameter:
  - the actual parameter is *evaluated* in the caller's context
  - then the value is copied into the function's formal parameter before the function begins executing

Pass by value

```c
int prod(int i, int j) {
  int k;
  k = i * j;
  return k;
}

int main() {
  int a, b, c, k;
  a = 1;
  b = 2;
  k = 0;
  c = prod(a, b); /* Call the function prod with input a and b */
  printf("a=%d, b=%d, c=%d, k=%d\n", a, b, c, k);
  return 0;
}
```

- variables within the function (`i`, `j`, `k`) are local
- they exist while the function is being executed
- they are removed when function exits

Pass by value vs. Pass by reference

C uses "pass by value":

- the actual parameter values are copied into local storage
- the caller and callee functions do not share any memory -- they each have their own copies

Two disadvantages by "pass by value":

- communication issue: modifications to that memory of callee's copy are not communicated back to the caller
  - the function's return value can communicate some information back to the caller
  - but not all problems can be solved with the return value
- Expensiveness: undesirable to copy the value from the caller to the callee if copying is expensive.

Pass by value vs. Pass by reference

"Pass by reference"

- the actual parameters are not copied but passed directly
- C does not support reference parameters automatically
- manually implemented using *pointers*

Example

```
void swap(int x, int y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}

/* Some caller code which calls swap()... */
  int a = 1;
  int b = 2;
  swap(a, b);
```

- the function `swap()` is supposed for swapping the values of `x` and `y`
- however, `swap()` does not affect the arguments `a` and `b` in the caller (due to "pass by value")
- when `swap()` exits, local memory (`x`, `y` and `temp`) disappears

Example

```c
void swap(int x, int y) {         /* NO does not work */
  int temp;
  temp = x;
  x = y;       /* these operations just change the local x, y, temp */
  y = temp;    /* -- nothing connects them back to the caller's a, b */
}

/* Some caller code which calls swap()... */
int a = 1;
int b = 2;
swap(a, b);
```

- the function `swap()` is supposed for swapping the values of `x` and `y`
- however, `swap()` does not affect the arguments `a` and `b` in the caller (due to "pass by value")
- when `swap()` exits, local memory (`x`, `y` and `temp`) disappears

Pointers (complex data type)

A pointer is a value which represents a reference to another value (the "pointee").

- Syntactically C uses the asterisk or "star" (`*`) to indicate a pointer.
- C defines pointer type based on the type of the pointee.
- A `char*` is a type of pointer which refers to a `char`.
- The `&` operator returns a pointer (to the argument to its right).

```c
int x;
int *xp;

x = 1;

printf("Value of x: %d\n", x); /* Value of x: 1 */
printf("Address of x: %p\n", &x); /* Address of x */

xp = &x; /* assign the address of x to the value of the pointer xp */
printf("Value of xp: %p\n", xp);
printf("Address of xp: %p\n", &xp); /* Address of xp */
```

More properties

- Pointer can be de-referenced by the unary * to the left of a pointer.

```
int x=1;
int *xp;

xp = &x;
/* assign the address of x to the value of the pointer xp */

printf("%d\n", *xp); /* de-referencing xp */
```

- Incrementing a pointer-to-an-int variable automatically adds to the pointer address the number of bytes used to hold an int (on that machine) (similarly for other floating types)
  - useful for array (will cover later)

Example

```c
int y;
int *p, *q;
int a, b;
int *i;

/* example 1 */
y = 1;
p = &y;   /* assign address of y to p */
q = p;    /* assign value of p to q
             (p and q share the same value: address of y) */
q = q + 2; /* incrementing a pointer */
printf("value of p: %p\n", p);
printf("value of q: %p\n", q);

/* de-referencing: what is the content p is pointing to? */
printf("pointing content of p: %d\n", *p);

/* example 2 */
a = 1;
i = &a;
b = *i;

printf("values of a and b are %d and %d\n", a, b);
printf("addresses of a and b are %p and %p\n", &a, &b);
```

Using pointers

When using pointers, there are two entities to keep track of:

- the pointer
- the "pointee": the memory it is pointing to

Three crucial things for a pointer/pointee relationship to work:

1. the pointer must be declared and allocated
2. the pointee must be declared and allocated
3. the pointer (step 1) must be initialized so that it points to the pointee (step 2)

Common pointer related error

- Declare and allocate the pointer (step 1)
- but forget step 2 and/or 3.

```
int *p;
*p = 13;  /* p does not point to an int yet */
          /* this just overwrites a random area in memory */
```

Correct way to use pointers

```
int* p;     /* (1) allocate the pointer */
int i;      /* (2) allocate pointee */
int j;      /* (2) allocate pointee */
p = &i;     /* (3) setup p to point to i */
*p = 42;    /* ok to use p since it's setup */

p = &j;     /* (3) setup p to point to a different int */
*p = 22;
```

Reference parameter technique

- To pass an object `x` as a reference parameter:
  - pass the pointer to `x` instead (i.e. the address of the memory of `x`)
  - we can change the value of `x` by changing the content of the pointer directly
  - operators `&` (referencing) and `*` (dereferencing) are typically used

```
void swap(int* x, int* y) {     /* params are int* instead of int */
  int temp;
  temp = *x;         /* use * to follow the pointer back to the
                        caller's memory */
  *x = *y;
  *y = temp;
}
/* Some caller code which calls swap()... */
int a = 1;
int b = 2;
swap(&a, &b);
```

  - the pointer will be copied ("pass by value"), but all we care is the address that it contains

Example

```c
#include<stdio.h>

void fun(int *p){
  printf("in fun...\n");
  *p = 2;
  printf("Value of p in fun: %p\n", p);
  printf("Address of p in fun: %p\n", &p);
  printf("out of fun...\n");
}

int main(){
  int x;
  int *xp;

  x = 1;
  xp = &x; /* assign the address of x to the value of the pointer xp */

  printf("Value of x: %d\n", x); /* Value of x: 1 */
  printf("Value of xp: %p\n", xp);
  printf("Address of xp: %p\n", &xp);

  fun(xp);
  printf("Value of x: %d\n", x); /* Value of x: 2 */

  return 0;
}
```

Another example

```
void Swap(int* a, int* b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

void IncrementAndSwap(int* x, int* y) {
  (*x)++;
  (*y)++;
  Swap(x, y);   /* don't need & here since a and b are already int*'s. */
}

int main() {
  int alice = 10;
  int bob = 20;

  Swap(&alice, &bob);
  /* at this point alice=20 and bob=10 */
  IncrementAndSwap(&alice, &bob);
  /* at this point alice=11 and bob=21 */

  return 0;
}
```