Programming in C

Biostatistics 615/815
Last Lecture

- Introduction to C Programming
  - Strongly typed language
  - Programs are a collection of functions

- The GNU Tool Collection

- Programming in a Windows Environment
Today

- Basic data types in C
- Common programming constructs
- Key library functions
Each variable is allocated a specific number of bytes (and bits) of memory.

Each variable's data type determines how these bits are interpreted.
Common Data Types in C

- **Integer data types**
  - `int`, `long`

- **Floating point data types**
  - `float`, `double`

- **Character types**
  - `char`

- **Pointers**
  - `int *`, `double *`, `char *`

- **User defined types**
  - `struct` *(short for structure)*

- **Empty data type**
  - `void`
Integers

- For most purposes the int type will do
  - unsigned int for strictly positive quantities

- Typically, store up to 31 or 63 digits
  - in base 2
  - plus one digit for sign
  - range is about -2.1 to 2.1 billion (32 bit)
C Operators

- Addition and subtraction:  +, -
- Multiplication:  *
- Division, Remainder:  /, %
- Increment, Decrement:  ++, --
Floating point

- Stored as exponent, mantissa and sign
  - Representation varies between machines

- Limited range and precision
Floating point data

- Stored in exponential notation
  - In base 2

- Has limited accuracy
  - Computing two similar quantities and evaluating their difference can be especially inaccurate

- Greater range than integer data
  - Exact for small integers
C Structures

- Use the `struct` keyword
- Must be defined before use
- Structure declaration
  - Name individual items
  - Define a type for each item
- Contents accessed with "." or "->" operators
C Structures

```c
typedef struct {
    int x;
    int y;
} point;

point origin;
origin.x = origin.y = 0;

point * ptr;
ptr->x = 0; ptr->y = 1;
```
Arrays

- Use \[ \] in declaration to specify size
- Indexes between $0 \ldots N-1$
- Text strings are arrays of characters
- Pointers are an alternative when array size depends on input …
Compound Statements

- C statements can be grouped with `{ }`

- Each compound statement starts with optional local variable declarations

- Individual statements separated by “;”
if ... else ...

```c
if (expression)
  statement1;
else
  statement2;
```

- **When** `expression` is true (or nonzero) `statement1` is executed; otherwise `statement2` is executed.
Example

```c
void Compare(int a, int b)
{
    if (a == b)
        printf("Values Match!\n");
    else
        printf("Values are different!\n");
}
```
**do ... while ...**

```plaintext
do 
  statement;
while (expression);
```

- **statement** is executed until **expression** evaluates to false (or zero).
- **statement** is executed is executed at least once.
Example

/* Calculate precision of double */
double precision()
{
  double e = 1.0, temp;

  do {
    e = e * 0.5;
    temp = 1.0 + e;
  } while (temp > 1.0);

  return e * 2.0;
}

while ...

while (expression)
    statement;

- statement is executed while expression evaluates to true.
- statement may never be executed.
Example

/* Calculate maximum integer */
double maximum_integer()
{
    int a = 2, b = 1, bits = 1;

    while (a > b)
    {
        b = a;
        a = a + a;
        bits++;
    }

    printf(“Looks like a %d-bit computer\n”, bits);

    return b;
}
for

for ( initialization; condition; increment)
    statement;

- **Executes** initialization.

- **While** condition is true:
  - Execute statement.
  - Evaluate increment.

- statement may never be executed.
Example

// Search through each item
for (i = start; i <= stop; i++)
  if (value == a[i])
    return i;
break and continue

- continue
  - Re-evaluates loop condition.
  - If not finished, start a new cycle.

- break
  - Stop looping early.
Important Library Functions

- `<stdio.h>`
  - Input and output

- `<stdlib.h>`
  - Basic random numbers and memory allocation

- `<time.h>`
  - Time and date information
Input / Output Functions

- `<stdio.h>`

- Default
  - `int printf(char * format, ...);`
  - `int scanf(char * format, ...);`

- File based functions
  - `FILE * fopen(char * filename, char * mode);`
  - `int fclose(FILE * file);`
  - `int fprintf(FILE * file, char * format, ...);`
  - `int fscanf(FILE * file, char * format, ...);`
\textbf{printf}

- Writes formatted output

- Format string controls how arguments are converted to text
  - Parameters are printed as specied in \% fields
    - \%[flags][width][precision]type
  - Otherwise, string is quoted
printf fields

- **Flags:**
  - “-” to left justify result
  - “+” to show sign in positive numbers

- **Width**
  - Minimum number of characters to print

- **Precision**
  - Number of digits after decimal (for floating point)
  - Maximum number of characters (for strings)

- **Type**
  - “s” for strings
  - “d” for integers, “x” to print hexadecimal integers
  - “f” for floating point, “e” for exponential notation, “g” for automatic
scanf

- Reads formatted input
- Format string defines input interpretation
  - Each `%%[type]` field is converted and stored
- Arguments should be addresses of variables where input is to be stored
scanf fields

- Field types
  - “s” for strings
  - “d” for int variables
  - “ld” for long variables
  - “f” for float variables
  - “lf” for double variables
Example

#include <stdio.h>

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

int main()
{
    int number;

    printf("Type a number:");
    scanf("%d", &number);
    printf("The square of %d is %d.\n", number, square(number));

    return 0;
}
Opening and closing files

- FILE * fopen(char * filename, char * type);
  - Opens file with *filename*
  - If type is “wt”, a text file is opened for writing
  - If type is “rt”, a text file is opened for reading
  - Types “rb” and “wb” are analogous for binary files
  - Returns NULL on failure

- int fclose(FILE * file);
  - Closes file
  - Returns 0 on success
Example

```c
#include <stdio.h>

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

int main()
{
    int number;
    FILE * output;

    printf("Type a number:");
    scanf("%d", &number);

    output = fopen("results.txt", "wt");
    fprintf(output, "The square of %d is %d\n", number, square(number));
    fclose(output);

    return 0;
}
```
Basic Random Numbers

- `<stdlib.h>`

- `int rand()`  
  - Sample a uniformly distributed random integer between 0 and RAND_MAX

- `void srand(int seed)`  
  - Select the sequence of random numbers specified by seed
Weighted Quick Union in C

// Initialize random generator
srand(1234);

// Generate M random connections
while (count++ < M)
{
  // Pick random elements to connect
  p = rand() % N;
  q = rand() % N;

  // FIND operation
  for (i = a[p]; a[i] != i; i = a[i] ) ;
  for (j = a[q]; a[j] != j; j = a[j] ) ;
  if (i == j) continue;

  // UNION operation
  if (weight[i] < weight[j])
    { a[i] = j; weight[j] += weight[i]; } 
  else
    { a[j] = i; weight[i] += weight[j]; }

  printf("%d %d is a new connection\n", p, q);
}


Dynamic Arrays ...

- Program must explicitly request memory
- Program should explicitly free memory
- Memory management provided by `<stdlib.h>`
- Pointers
  - Store a memory address
  - `int *`, `double *`, `char *`
Pointers in C

- Declared with data type followed by *

- Can be initialized by…
  - Calling `malloc()` to request new memory block
  - Retrieving address of existing variable (`&var`)

- Accessed using …
  - `pointer` to retrieve memory address
  - `*pointer` to retrieve first element
  - `pointer[i]` to retrieve $i^{th}$ element
Basic Memory Management

- `<stdlib.h>`
- `void * malloc(size_t bytes)`
  - Allocates a block of memory
  - Required amount specified in bytes
  - Pointer can be converted to appropriate type

- `void free(void * pointer)`
  - Releases memory

- `sizeof(type)`
  - Returns size of data type in bytes
Example

- Static allocation

  ```c
  #define N 1000
  int data[N];
  ```

- Dynamic allocation

  ```c
  int N, * data;

  printf("How many data points?\n");
  scanf("%d", &N);
  data = malloc(sizeof(int) * N);
  ```
Time and Date Functions

- `<time.h>`

- `time_t time(time_t * timer);`
  - Returns the number of seconds since 1/1/1970.

- `char * ctime(time_t * timer);`
  - Formats the current date as a string.
```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t start, stop;
    int seconds;

    start = time(NULL);
    // Carry out some computation ...
    stop = time(NULL);
    seconds = stop - start;

    printf("Run completed in %d seconds on %s\n", stop, ctime(stop));
    return 0;
}
```
Today

- Basic data types in C
  - Limitations of integer and floating point types

- Common programming constructs

- Key library functions
  - `printf`, `scanf`, `fopen`, `fclose`
  - `rand`, `srand`
  - `malloc`, `free`