Final Review

CS 2060

Prof. Jonathan Ventura

Variables

- Variables are statically typed.
- Variables must be defined before they are used.
- You only specify the type name when you define the variable.

```
int a, b, c;
float d, e, f;
char letter;

// Variable assignment:
a = 5;

// Variable expressions:
b = a + 10;
```

Variables

- Variables are not initialized to any particular value.
- Variables should be initialized before use.

```
int a, b, c;

// bad: a is used uninitialized:
b = a + 10;
```

Functions

- Most C code is organized into functions.
- A function consists of
  - a return type,
  - name
  - argument list
  - and body.

```
float square( float a ) {
    return a * a;
}
```

Functions

- You call a function using its name and arguments.

```
float a = 2;
float asq;

asq = square(a);
```

Functions

- A function *signature* must be defined before it can be called.
- Or, the function signature **and** function body can be given.

```
// Function signatures:
int foo( float a );
void bar( int b, char c );

// Function definitions:
int foo( float a ) {
    return floor(a*5);
}
```

Functions

- The return type is `void` if the function does not return anything.

```
// Returns an integer:
int foo( float a );
// No return value:
void bar( int b, char c );
```

Functions

- Your program starts and ends in the `main()` function.

```
int foo( float a ) {
    return floor(a*5);
}

int main() {
    // Start of program
    float a = 10;
    int b = foo( a );

    // End of program
    return 0;
}
```

Variable scope

- *Local variables...*
  - are defined inside a function.
  - only exist for the duration of a function call.
  - can only be accessed from inside the function.

```
int foo( float a ) {
    return floor(a*5);
}

int main() {
    // a and b are local variables
    float a = 10;
    int b = foo( a );

    return 0;
}
```

Variable scope

- *Global variables...*
  - are defined outside any function.
  - exist for the duration of the program.
  - can be accessed from anywhere in the code.

```
// a is a global variable
float a = 10;

int main() {
    // b is a local variable
    int b = floor( a*5 );

    return 0;
}
```

Expressions

- Arithmetic expressions are formed using chains of operators.
- The result of the expression depends on the operator *precedence*.
  1. Parentheses ()
  2. Multiplication, division, modulo */%
  3. Addition, subtraction +-

```
int b = 3, c = 4, d = 5;

a = b + c * d;
a = ( b + c ) * d;
a = b + ( c * d );
```

Order of Arithmetic Operations

Operator(s)	Operation(s)	Order of evaluation (precedence)
( )	Parentheses	Evaluated first, inner to outer.
*	Multiplication	Evaluated second, left to right.
/	Division	
%	Modulo	
+	Addition	Evaluated third, left to right.
-	Subtraction	
=	Assignment	Evaluated last.

Table: Precedence of arithmetic operators in C.

Increment and decrement operators

- Increment and decrement operators:

  `a++` or `++a` will add one to `a`.
  `a--` or `--a` will subtract one from `a`.

- The difference is in what the expression evaluates to.

  Postfix: `a++`

  will increment `a` **after** returning the (old) value of `a`.

  Prefix: `++a`

  will increment `a` **before** returning the (new) value of `a`.

Assignment operators

- Assignment operators combine arithmetic and assignment:

```
a += 5;
a -= 5;
a *= 5;
a /= 5;
```

The above statements are equivalent to:

```
a = a + 5;
a = a - 5;
a = a * 5;
a = a / 5;
```

Comparison operators

- Comparison operators are used for Boolean (true/false) tests:
  - == tests equality
  - != tests inequality
  - >= tests greater-than-or-equal-to
  - <= tests less-than-or-equal-to

Comparison operators

```
char getGrade( int score ) {
    if ( score >= 90 ) return 'A';
    else if ( score >= 80 ) return 'B';
    else if ( score >= 70 ) return 'C';
    else if ( score >= 60 ) return 'D';
    else return 'F';
}
```

Logical operators

- Logical operators are used to form Boolean expressions.
  - `&&` **(logical AND)**
  - `||` **(logical OR)**
  - `!` **(logical NOT)**
- Their functions are as follows:
  - `(a && b)` is true only if *both* `a` and `b` are true.
  - `(a || b)` is true if *either* `a` or `b` is true.
  - `(!a)` is true if `a` is false.

Comparison operators

```
char getPassFail( int score, int attendance ) {
    if ( score >= 60 && attendance >= 50 ) return 'P';
    else return 'F';
}
```

Array syntax

- Arrays are continuous blocks of data of the same type.
- By default, arrays are uninitialized (data could be anything).

```
int numbers[100];
float values[10];
char string[20];
```

Array syntax

- A curly brackets initializer sets the elements of the array.

```
int numbers[5] = { 1, 2, 3, 4, 5 };
```

Array syntax

- Arrays are indexed using square brackets.
- The first element has index 0.

```
numbers[0] = 10;
```

`for` loops sequence of execution

The syntax of a for loop is as follows:

```
for ( /*initialization*/ ; /*condition*/ ; /*increment*/ )
{
    /* statements */
}
```

1. Execute *initialization*.

   Loop:

   1. Test *condition* – stop if false
   2. Execute *statements*
   3. Execute *increment*

Iterating through arrays

- `for` loops are ideal for iterating over arrays.

```
int numbers[10];
for ( int i = 0; i < 10; i++ ) numbers[i] = i;
```

Iterating through arrays

- Accessing elements 0,2,4,...:

```
int numbers[10] = { 0 };
for ( int i = 0; i < 10; i += 2 ) numbers[i] = i;
```

while and do...while loops

Other kinds of loops:

```
while ( /*condition*/ )
{
    /* statements */
}

do
{
    /* statements */
} while ( /*condition*/ )
```

Passing arrays to functions

- Use empty square brackets to indicate an array in function arguments.
- C doesn't keep track of the size of the array when passed to a function.
- You need a separate variable for the size of the array.

```
float sumArray( float array[], int size ) {
  float sum = 0;
  for ( int i = 0; i < size; i++ ) sum += array[i];
  return sum;
}
```

Passing arrays to functions

- Pass an array to a function using its name only.

```
int main() {
  float my_array[3] = { -1.f, 0.f, 1.f };
  float my_sum;

  my_sum = sumArray(my_array,3);
}
```

Standard C math library functions in math.h

Function	Description
`sqrt(x)`	$\sqrt{x}$
`exp(x)`	e^x
`log(x)`	$\ln(x)$
`log10(x)`	$\log_{10}(x)$
`abs(x)`	$\lvert x \rvert$ (`int`)
`fabs(x)`	$\lvert x \rvert$ (`double`)
`ceil(x)`	$\lceil x \rceil$
`floor(x)`	$\lfloor x \rfloor$
`pow(x,y)`	x^y
`sin(x),cos(x),tan(x)`	$\sin(x), \cos(x), \tan(x)$

Table: Commonly used math library functions

Random number generation

- The `rand()` function (defined in `stdlib.h`) generates random integers between 0 and `RAND_MAX`.

```
// Get a random integer
int i = rand();
// Convert to floating point in range [0,1]
float f = (float) i / (float) RAND_MAX;
```

Random number generation

- Use modulo and addition to generate random numbers in a specified range.

```
// Get a random integer in range [1 100]
int i = rand()%100 + 1;
```

Characters in C

- The `char` type is used to represent characters.
  - `char` is an 8-bit integer (range 0 to 255).
- A character is represented by its ASCII code:
  - A = 65, B = 66, ...
  - a = 97, b = 98, ...

Characters in C

- Use single quotes around a character to get its ASCII representation.

```
char letter_a = 'a';
char letter_A = 'A';
```

- Use %c to read or write a character using scanf or printf.

switch statements

- The switch statement selects among many options.
- The value of the control variable determines which option is selected.

```
switch ( getchar() ) {
    case 'r':
        num_red++;
        break;

    case 'b':
        num_blue++;
        break;

    default:
        num_other++;
}
```

switch statements

- The control variable can only be a basic integer type like `char` or `int`.
- The cases only test equality.

```
switch ( menu_selection ) {
    case 1:
        menu_option_1();
        break;

    case 2:
        menu_option_2();
        break;

    default:
        puts("no such option");
}
```

Storing strings in character arrays

- Strings are character arrays:

```
char string1[] = "first";
```

- In C, strings end with a special *string-termination character* known as a **null character**.
  - C strings are called **null-terminated strings**.
- The above is equivalent to:

```
char string1[] = {'f','i','r','s','t','\0'};
```

Copying a string

- Use `strcpy` to copy a string:

```
char *strcpy( char *to, const char *from );
```

- Remember that the first argument is the destination and the second argument is the source – this is typical in the C Standard Library.

Comparing strings

- `strcmp` tests string equality:

```
int strcmp( char *s1, const char *s2 );
```

- It returns 0 if the strings are equal.

String length

- `strlen` finds the length of a string.

```
size_t strlen( const char *str );
```

- The length does not include the null terminator.

structs

- A *struct* contains variables of possibly different types:

```
struct Employee {
    char first_name[20];
    char last_name[20];
    unsigned int age;
    char gender;
    double hourly_salary;
};
```

- A `struct` is defined outside of any function.
- Don't forget the semi-colon at the end.

C Structures: `struct`

- The type name includes the word `struct`.
- Member variables are accessed using the `.` operator.

```
int main() {
    struct Employee employee1;
    strcpy( employee1.first_name, "Jane" );
    strcpy( employee1.last_name, "Doe" );
    employee1.age = 22;
    employee1.gender = 'f';
    employee1.hourly_salary = 20;
}
```

Pointers

- A pointer is a variable whose value is a *memory address*.
- The variable does not contain meaningful data itself, but instead **points** to where data is located in memory.

- Given the pointer (a memory address), we **dereference** the pointer to read/write the value stored at that address.

Defining pointers

- Every data type has an associated pointer type, like `int *`:

```
int count = 10;
int *countPtr = &count;
```

- The **reference** operator `&` gets a pointer to a variable.
- Variable `countPtr` contains the memory address of `count`.

Dereferencing pointers

- The **dereference** operator * gets the variable pointed to by a pointer:

```
int count = 10;
int *countPtr = &count;
*countPtr = 20;
```

- count will equal 20 now.

Passing arguments by reference with pointers

- With pointers we can implement pass-by-reference:

```
void square( float *valuePtr ) {
    *valuePtr = (*valuePtr) * (*valuePtr);
}
```

- The function modifies the value pointed to by `valuePtr`.

Pointer arithmetic operators

- Incrementing an `int *` pointer will move it to the next `int` in memory.

```
int data[10];
int *dataPtr = data; // points to data[0]
*dataPtr = 0;
dataPtr++;  // points to data[1]
*dataPtr = 10;
dataPtr++;  // points to data[2]
```

- The first two elements of `data` will be 0 and 10.

Pointer arithmetic example

- To iterate over an array using a pointer:

```
int array[5] = { 0, 1, 2, 3, 4 };
for ( int *ptr = array; ptr != array+5; ptr++ )
{
    printf("%d\n", (*ptr) );
}
```

Pointers to structs

- The -> operator accesses `struct` members through a pointer.

```
void printRoom( struct Room *room )
{
    printf( "%s\n", room->name );
    printf(" %s\n", room->description );
}
```

Dynamic memory allocation

- `malloc` allocates memory and returns a pointer to it.
- `free` frees the memory when we are done with it.

```
int *data = malloc( 10 * sizeof(int) );
free( data );
```

- `data` is just like an array of 10 `int`s.

Dynamic memory allocation example

- Use `malloc` to allocate space for a `struct`:

```
struct Point { int x, int y };

struct Point * makePoint( int x, int y )
{
    // allocate struct
    struct Point *pt = malloc( sizeof(struct Point) );

    // initialize struct
    pt->x = x;  pt->y = y;

    // return pointer
    return pt;
}
```

Function Pointers

- A **function pointer** points to a function.

```
void runTask( void (*callbackFn)( int status ) ) {
    for ( int i = 0; i < 100; i++ ) {
        // .. do some computation ...
        (*callbackFn)( i );
    }
}

void printStatus( int status ) {
    printf( "status: %d \% done\n", status );
}

// ....
    runTask( printStatus );
```

- The `runTask` function takes a function pointer as an argument.
- During a long process, it calls the callback to print out status information.

Files and Streams

- Files are used for long-term storage of data
  - (on a hard drive rather than in memory).
- The data in a file is accessible through file access functions such as `fprintf` and `fscanf` or `fread` and `fwrite`.

FILE structure

- Open a file with `fopen` and close it with `fclose`:

```
FILE *f = fopen( "hello.txt", "r" );
fclose( f );
```

- `fopen` returns a `FILE *` pointer.
- The first argument is the file name.
- The second argument is the *file open mode*.

File open modes

- The most common file open modes:

Mode	Description	Notes
`"r"`	Read-only	Opens existing file
`"w"`	Write-only	Creates new file, deleting old file if it exists
`"a"`	Append	Begins with file pointer at end of file
`"r+"`	Read and write	Opens existing file
`"w+"`	Read and write	Creates new file, deleting old file if it exists

Reading from a file

- Use `fscanf` for formatted input from a file:

```
// Open hello.txt as read-only
FILE *f = fopen("hello.txt","r");

// Read five integers from the file
int a[5];
for ( int i = 0; i < 5; i++ ) {
    fscanf( f, "%d", &a[i] );
}

// Close the file
fclose(f);
```

Formatted print

- `fprintf` writes formatted output to a file:

```
// Open output.txt
FILE *f = fopen("output.txt","w");

// Write out some text
fprintf( f, "Hello, world!\n" );
fprintf( f, "2 + 5 = %d\n", 2 + 5 );

// Close the file
fclose(f);
```

Reading integers stored as binary

- `fread` reads bytes from a file:

```
// Open the file for reading
FILE *f = fopen("integers.dat","r");

// Read 10 integers from the file
int values[10];
fread( values, sizeof(int), 10, f );

// Close the file
fclose(f);
```

- `fwrite` is similar.

Random Aaccess files

- A *random access file* contains a sequence of `structs`.

Starting byte #:	0	100	200	300	...
Record #:	0	1	2	3	...

Table: A random access file with 100-byte records

Random access file

```
/** ClientData
 * Stores data for a bank client
 */
typedef struct {
    unsigned int account_num;    // account number
    char last_name[15];          // account last name
    char first_name[10];         // account first name
    double balance;              // account balance
} ClientData;
```

Updating a random access file

```
int main() {
    // Open file
    FILE *f = fopen( "accounts.dat", "r+" );

    // Move file pointer to tenth record
    fseek( f, 9*sizeof(ClientData), SEEK_SET );

    // Read requested record
    ClientData client_data;
    fread( &client_data, sizeof(ClientData), 1, f );
```

Reading a random access file

```
    // Zero out balance
    client_data.balance = 0;

    // Move file pointer back to tenth record
    fseek( f, 9*sizeof(ClientData), SEEK_SET );

    // Update record in file
    fwrite( &client_data, sizeof(ClientData), 1, f );

    // Close file
    fclose( f );
}
```