C Programming

Mark Allen Weiss
Copyright 2000

Outline

- **Overview of C**
- **Functions**
- **C-style Pointers**
- **Preprocessor**
- **Arrays**
- **Strings**
- **structs**
- **I/O**
- **Mixing C and C++**

C Basics

- **High-level assembler language**
  - **basic constructs of high-level languages**
  - **ports to many machines**
  - **allows access to system resources ala assembler**
- **1970s language philosophy**
  - **assumes the programmer knows best**
    - **relatively few compiler checks**
    - **relatively few runtime checks**
  - **loose type checking**
  - **not object-oriented or even object-based**

Versions Of C

- **Original K&R C (1970s)**
  - **Spec is ambiguous in come places**
- **ANSI C (1988)**
  - **Attempts to clean up original spec**
  - **Attempts to codify some programming tricks**
  - **Adds notion of the function prototype**
  - **The version to program to**
- **Non-standard C**
  - **many compilers add features; can turn off extensions with compiler options**
- **C99**

Similarity

- **Same set of primitive types**
  - **short, int, long, unsigned, signed, float, double, char**
  - **no boolean in ANSI C (0 is false, non-zero is true)**
- **Same set of operators**
  - **arithmetic, relational, equality, logical, bitwise, ?:, and assignment operators all the same**
- **Same types of loops and statements**
  - **for, while, do, switch, break, continue, return**

What ANSI C is Missing vs. C++

- **Classes and object-based programming**
- **first-class array and string types**
- **Strong(er) type checking**
- **Reference variables and call-by-reference**
- **Function and Operator overloading**
- **templates**
- **exceptions**
- **default parameters**
- **various nice coding benefits present in C++**
- **no `//` comments**

Printing to Terminal

- Use `printf` to print a string.
- Inside string use % escapes to add parameters:
  - int %d
  - reals %f
  - characters %c
  - other strings %s
- `printf` is not type-safe
- Example

```
int x = 37;
double y = 56.56;
printf( "The int x is %d. The double y is %f", x, y );
```

Functions

- Same ideas as in C++
- Variables must be declared at the start of the function
  - once you have a non-declaration statement, cannot declare any more local variables
- No overloading allowed
- No inline declarations allowed
- All parameters are passed call-by-value: no exceptions
- Prototypes, ala C++ are allowed but not required

Simulating Call By Reference

- Same idea as Java, actually!
- Pass a pointer as a parameter to a function
  - cannot change value of the pointer
  - can change state of the object being pointed at
- Function declares that it is receiving a pointer
- Dereference pointer with `*` (just like C++)
- Pass a pointer as the actual argument
  - If you have an object, can get its address with address-of-operator (&)

Swapping Example

```
#include <stdio.h>

void swap( int * x, int * y ) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main( void ) {
  int a = 5, b = 7;
  swap( &a, &b );     /* must pass the address */
  printf( "%d %d\n", a, b );
  return 0;
}
```

Reading From Terminal

- **Use `scanf` to read (return value is # items read)**
  - **Inside controlling string use % escapes for each parameter:**
    - **ints %d, %ld, etc.**
    - **reals %f, %lf, etc.**
  - **Pass addresses of variables to be filled in**
- **`scanf` is not type-safe**
- **Example**

```
int x;
double y;
int itemsRead;
itemsRead = scanf( "%d %lf", &x, &y );
```

Preprocessor Macros

- **Preprocessor directives begin with #**
  - **`#include`, `#define`, `#ifndef`, `#else`, `#endif`, `#undef`, `#if`, `#else`, etc.**
  - **long lines can be continued with `\`**
- **Same set as in C++, but used much more often in C**
- **Preprocessor macros perform textual substitution (logically before) program is compiled**

Simple Textual Substitution

```
#define MAX 50
#define SUMAB a + b
```

this text

```
if( x == MAX )
  c = SUMAB * SUBAB;
```

becomes

```
if( x == 50 )
  c = a + b * a + b;
```

- **Moral of the story: always overparenthesize**

Trivial Functions

- **Used in C for speed**

```
int absoluteValue( int x )
{
  return x >= 0 ? x : -x;
}
```

- **Above function is trivial, but overhead of function call could be significant**
  - 70s compilers not very good at that
  - tendency of programmers to inline functions themselves yields bad software engineering
  - modern compilers are very good at inlining; even so, often see macros used instead of functions

Parameterized Macros

- **Macro expansion: textual substitution, with actual arguments directly substituted for formal parameters**

```
#define absoluteValue(x) ( (x)>=0 ? (x) : -(x) )
y = absoluteValue(a-3);
z = absoluteValue(--n);
```

- **becomes**

```
y = ( (a-3)>=0 ? (a-3) : -(a-3) );
z = ( (--n)>=0 ? (--n) : -(--n) );
```

- **Parameterized macros ARE NOT semantically equivalent to function calls.**
  - arguments may be evaluated multiple times.

Arrays

- **Not first-class array type in C**

```
int a[ 20 ];
int *b;  // allocate some memory elsewhere
```

- **Like C++**
  - **array value is a pointer to memory**
  - **indexing starts at zero**
  - **no bounds check**
  - **array value does not have any idea of how large the array is, except in case where array is allocated using `[]` syntax**
  - **memory is not reclaimed, except in case where array is allocated using `[]` syntax**

Passing Arrays

- **Use either `[]` or `*` in function declaration**
  - **`[]` follows type name**
- **Use `const` to indicate that state of the array will not change**
- **Pass the array (i.e. the pointer to the start)**
- **Probably have to pass the number of items too**

```
/* Declarations */
void printItems( const int *arr, int n );
void initialize( int arr[], int n );
```

- **Size in `[]` is ignored**
- **Cannot return array objects -- only pointers**

Allocating and Deallocating

- **Must use `malloc` to allocate array if its size is not known at compile time or you want to change size as program runs**
  - **not type-safe; returns a `void*`**
  - **returns `NULL` if no memory**
- **Must use `free` to deallocate any memory that was allocated by `malloc`**
- **Can use `realloc` to increase amount of memory allocated for an array; it obtains more memory, copies items over, and frees original**
  - **Parameters to `malloc` and `realloc` are #bytes of memory to obtain**

Sample Reallocating Code

```
/* Returns a pointer to the data */
/* itemsRead is set by reference to #items read */
int * getInts( int * itemsRead ) {
  int numRead = 0, arraySize = 5, inputVal;
  int *array = malloc( sizeof( int ) * arraySize );
  if( array == NULL )
    return NULL;
  printf( "Enter any number of integers: " );
  while( scanf( "%d", &inputVal ) == 1 ) {
    if( numRead == arraySize ) {  /* Array Doubling Code */
      arraySize *= 2;
      array = realloc( array, sizeof( int ) * arraySize );
      if( array == NULL )
        return NULL;
    }
    array[ numRead++ ] = inputVal;
  }
  *itemsRead = numRead;
  return realloc( array, sizeof( int ) * numRead );
}
```

Multidimensional Arrays

- **Very messy to do fancy stuff**
- **If you know dimensions, it is easy**

```
int x[4][7];  // declares 4x7 array
```

- **Formal parameters must include all dimensions except the first may be omitted**

```
void print( int y[][7], int numRows );
```

Pointer Math

- **Given a pointer `p`, `++p` changes the value of the pointer to point at an object stored one unit higher in memory**
- **If `p` is pointing at an object in an array,**
  - `++p` points at the next object
  - `p+k` points at the object k away
  - `p1-p2` is the separation distance of two objects in an array
- **Gives a 70s style idiom for traversing an array**
- **Most optimizing compilers make this idiom obsolete, but you will see it anyway**

Two ways of initializing an array

```
void initialize1( int arr[], int n )
{
  int i;
  for( i = 0; i < n; i++ )
    arr[ i ] = 0;
}

void initialize2( int arr[], int n )
{
  int *endMarker = arr + n;
  int *p = arr;
  while( p != endMarker )
    *p++ = 0;
}
```

Characters

- **Use `putchar` to print a single character**
- **`getchar` to read a single character**
  - **returns an `int`, `EOF` if end of file**
- **`<ctype.h>` contains various character testing routines such as `isdigit`, `isalpha`, etc. These are all macros. Also contains `toupper` and `tolower`.**

Strings

- **Represented as an array of `char`**
- **After last character in string, there is a null terminator `'\0'`**
  - **placed there automatically for constants**
  - **placed there automatically by library routines**
- **String library routines:**
  - **`strlen`: returns length of string (`'\0'` not included)**
  - **`strcmp`: compares two null-terminated strings; same semantics as Java's `compareTo` function**
  - **`strcpy`: copies second parameter into first; must be enough array space in first parameter or you can get in trouble**

The Problem With C Strings

- **Very common to have buffer overflow problems**
  - **array allocated for string is not large enough**
  - **need to remember null terminator**
  - **cannot assume limits on input line length, etc.**
- **Common error**

```
char *copy;
char orig[] = "hello";  // six character array
strcpy( copy, orig );   // crashes: no memory
```

`sprintf` and `sscanf`

- **Similar to stringstreams in C++**
- **First parameter to `sprintf` is a string in which to write (instead of file or terminal)**
- **First parameter to `sscanf` is a string to parse**

```
int x;
double y;
int items = sscanf( "37 46.9", &x, &y );
```

Arrays of Strings

- **Typically declared as `char *arr[]`**

```
const char *ERRORS[ ] = { "Out of memory",
  "Input value out of range", "Format error",
  "Premature end of input" };
```

- **Example is parameter to `main`**

```
#include <stdio.h>
int main( int argc, char *argv[], char *envp[] )
{
  int j;
  printf( "ENVIRONMENT\n" );
  for( j = 0; envp[j] != NULL; j++ )
      printf( "%s\n", envp[ j ] );
}
```

C Pointer Dangers

- **Returning a pointer to a static function variable**
  - **Must use value of object being pointed at prior to next call to the function or it is overwritten**
- **Returning a pointer to a local variable**
  - **Always wrong; local variable likely to be destroyed and you have a stale pointer**
- **Returning a pointer to a dynamically allocated local object**
  - **You must take responsibility for calling `free` or you have a potential memory leak**

Structures (`structs`)

- **Precursor to C++ classes**
  - **no methods or constructors**
  - **no private -- everything is public**
  - **have to say `struct` when using type**
- **K&R C: cannot pass or return a struct**
- **ANSI C: OK to pass or return a struct**
  - **but if `struct` is large, this is not a good idea, since it involves a copy**
- **Structs are almost never passed to or returned from functions. Instead pointers to structs are used**

Example: time.h

```
struct  tm {
  int     tm_sec;    /* seconds after the minute (0- 61) */
  int     tm_min;    /* minutes after the hour   (0- 59) */
  int     tm_hour;   /* hours after midnight     (0- 23) */
  int     tm_mday;   /* day of the month         (1- 31) */
  int     tm_mon;    /* month since January      (0- 11) */
  int     tm_year;   /* years since 1900         (0-   ) */
  int     tm_wday;   /* days since Sunday        (0-  6) */
  int     tm_yday;   /* days since January 1     (0-365) */
  int     tm_isdst;  /* daylight savings time flag       */
};
typedef long    time_t;

/* Some functions */
extern time_t mktime(struct tm *);
extern char *asctime(const struct tm *);
```

Illustration of Passing Structs

```
/* Find all Friday The 13th birthdays for person born Nov 13, 1973 */
#include <time.h>
#include <stdio.h>
int main( void ) {
  const int FRIDAY = 6 - 1;          /* Sunday Is 0, etc... */
  struct tm theTime = { 0 };         /* Set all fields To 0 */
  int year;
  theTime.tm_mon = 11 - 1;           /* January is 0, etc... */
  theTime.tm_mday = 13;              /* 13th day of the month */
  for( year = 1973; year < 2073; year++ ) {
    theTime.tm_year = year - 1900;   /* 1900 is 0, etc... */
    if( mktime( &theTime ) == -1 ) {
      printf( "mktime failed in %d\n", year );
      break;
    }
    if( theTime.tm_wday == FRIDAY )
      printf( "%s", asctime( &theTime ) );
  }
  return 0;
}
```

Pointers to Functions

- **Can pass functions as parameter to other function**
  - **technically you pass a pointer to the function**
  - **syntax can look clumsy, but in ANSI C can avoid clumsy syntax**

```
double derivative( double f( double ), double x ) {
  double delta = x / 1000000;
  return ( f( x + delta ) - f( x ) ) / delta;
}

int main( void ) {
  printf( "Deriv is %f\n", derivative( sqrt, 1.0 ) );
}
```

Equivalent Code With Pointers

```
double derivative( double ( *f) ( double ), double x ) {
  double delta = x / 1000000;
  return ( (*f)( x + delta ) - (*f)( x ) ) / delta;
}

int main( void ) {
  printf( "Deriv is %f\n", derivative( sqrt, 1.0 ) );
}
```

Pointers to Functions as Fields

```
void help( void );
void quit( void );

struct Command {
  char *command;
  void ( *func )( void );
};

struct Command theCommands[ ] = {
  "exit", quit,
  "help", help,
  "quit", quit,
  /* etc. */
  NULL, NULL             /* Place last; No match */
};
```

Using the Pointers

```
void doCommand( const char *comm ) {
  struct Command *ptr;
  for( ptr = theCommands; ptr->command != NULL; ptr++ )
    if( strcmp( comm, ptr->command ) == 0 ) {
      ( *ptr->func )( );
      return;
    }
  printf( "Error: unrecognized command\n" );
}
void help( ) {
  printf( "Here's my help!\n" );
}
void quit( ) {
  exit( 0 );
}
```

qsort

- **Generic sorting algorithm**

```
void qsort( void *arr, int n, int itemSize,
               int cmp( const void *, const void * ) );
```

- **Typical of how generic stuff is done in C**
- **Example: sorting array of ints:**

```
int arr[] = { 3, 5, 1, 2, 6 };
qsort( arr, 5, sizeof( int ), intCmp )
```

where comparison function is

```
int intCmp( const void *lhs, const void *rhs )
{
  int lhint = *(const int *)lhs;
  int rhint = *(const int *)rhs;
  return lhint < rhint ? -1 : lhint > rhint;
}
```

Files

- Associate a stream with a file
- Stream represented by a FILE object, defined in stdio.h
  - these objects are passed using pointers
  - Various routines to read/write; all start with f
  - can be opened for reading or writing or both
- Standard streams are `stdin`, `stdout`, and `stderr`

Important Routines

- `fopen` and `fclose`
  - open with a mode such as "r" or "w"
  - fopen returns `FILE *`; `NULL` if error
- `fprintf` and `fscanf`
  - work just like printf and scanf
  - first parameter is a FILE *
- `fgetc` and `fputc`
  - work like getchar and putchar
  - last parameter is a FILE *
  - often implemented as a preprocessor macro

More Routines

- `fgets` and `fputs`
  - Reads/writes strings
  - fgets reads a line or input, with a limit on number of characters
    - newline included in string if it was read
    - make sure you have enough space for newline and '\0'
- `feof`
  - returns true if read has already failed due to EOF
- `fread` and `fwrite`
  - Allows reading of binary data into a struct or array
- `fseek` and `ftell`
  - Allows random access of files

Example: File Copy: part 1

```
int copy( const char *destFile, const char *sourceFile ) {
  int charsCounted = 0, ch;
  FILE *sfp, *dfp;

  if( strcmp( sourceFile, destFile ) == 0 ) {
    printf( "Cannot copy to self\n" );
    return -1;
  }

  if( ( sfp = fopen( sourceFile, "r" ) ) == NULL ) {
    printf( "Cannot open input file %s\n", sourceFile );
    return -1;
  }

  if( ( dfp = fopen( destFile, "w" ) ) == NULL ) {
    printf( "Cannot open output file %s\n", destFile );
    fclose( sfp ); return -1;
  }
```

Part 2: Character at a Time

```
  while( ( ch = getc( sfp ) ) != EOF )
    if( putc( ch, dfp ) == EOF )
    {
      printf( "Unexpected error during write.\n" );
      break;
    }
    else
      charsCounted++;

  fclose( sfp );
  fclose( dfp );
  return charsCounted;
}
```

File Copy: Line at a Time

```
#define MAX_LINE_LEN 256
int copy( const char *destFile, const char *sourceFile )
{
  int charsCounted = 0;
  char oneLine[ MAX_LINE_LEN + 2 ];
  FILE *sfp, *dfp;
  // ... same start

  while( ( fgets( oneLine, MAX_LINE_LEN, sfp ) ) != NULL )
    if( fputs( oneLine, dfp ) < 0 ) {
      printf( "Unexpected error during write.\n" );
      break;
    }
    else
      charsCounted += strlen( oneLine );

  // ... same finish
}
```

Example: Printing Last Chars in File

```
void printLastChars( const char *fileName, int howMany ) {
  FILE *fp;
  char *buffer = NULL;
  int charsRead, fileSize;

  buffer = malloc( howMany );   /* error check omitted */
  fp = fopen( fileName, "rb" ); /* error check omitted */

  fseek( fp, 0, SEEK_END );     /* go to end */
  fileSize = ftell( fp );       /* get position */
  if( fileSize < howMany )
    howMany = fileSize;

  fseek( fp, - howMany, SEEK_END );
  charsRead = fread( buffer, 1, howMany, fp );
  fwrite( buffer, 1, charsRead, stdout );
  fclose( fp );
  free( buffer );
}
```

Should I Use C

- **Good reasons to not write C code**
  - **have to manage your own memory for arrays and strings**
  - **variables must be declared at top of function**
  - **I/O is much messier than C**
  - **no overloading**
  - **no classes or templates**
  - **no type checking**
- **Reason to use C**
  - **might be faster**
  - **might need to interface to C library**

Calling C From C++

- **Best solution: write most of your code in C++**
- **Most C and C++ compilers are the same, so little speed benefits**
- **From C++, can access C routines if magic incantation provided:**
  - **extern "C" ...**
  - **may need to change search path to find include and library files**
  - **entire C library is part of C++**

Example

- **Suppose there is a C routine**

```
void foo( SomeObj *obj );
```

- **From C++:**

```
extern "C" void foo( SomeObj *obj );
int main( )
{
  SomeObj *p = ...;
  ...
  foo( p );
}
```

Using C in Your C++ Code

- **I/O using `FILE *` is generally much faster than using `ifstream` and `ofstream`**
- **Direct access of characters in a `string` might be faster using `char*`. Can get `char*` from a `string` using `c_str` member function**
  - or may need to use `char*` to save space in some cases
- **Don't:**
  - mix C and C++ streams
  - mix `new`/`delete` and `malloc`/`free`
  - forget that you probably have to pass structs using pointers or addresses

Summary

- **With C you lose many of C++ conveniences such as**
  - strings/vectors
  - type safety
  - ease of variable declarations
- **C is not object-oriented, or even object-based**
- **If you have to write C, you will miss C++**
- **If possible, write C++, and minimize use of C-style logic**