C Programming for the C++ Programmer

An Introduction

Overview

- History of C
- No class!
- Our first C program
- Compiling and running
- Makefiles
- Strings
- Standard I/O
- Structs
- Dynamic Memory
- Parameter Passing
- Overloading
- Casting
- Includes
- Implicit declarations
- Function Pointers
- Command line arguments

Language History

- Developed in early 70's
- Specified in 1978 in The C Programming Language by Kernighan and Ritchie.
- Modified and standardized in 1989.
    - Most important feature was specifying types in the parameter lists.
    - The second (current) edition of K&R describes this version.
- C99 added some nice flexibility, in particular being free to define a variable anywhere before its use.
- The C11 standard for the language is available at:
    - http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
    - Largely a clean-up. Added a threading library.

No Class!

- No class, only struct
- What’s the impact?
- No methods. No constructors. No destructors.
- No private / protected.
- No string, iostream, ifstream, vector …
- No overloading of operators:
- << and >> have nothing to do with I/O!
- No templates, STL,…
- No inheritance

First C Program

```c
/*
  hello.c
*/

#include <stdio.h>

int main() {
    puts("Hello Poly!");
    return 0;
}
```

- Comments: /* */. C99 also accepts //
- Even main should explicitly return a value
- The standard I/O library is stdio.h. Note the ".h"
- puts prints a string to standard output, appending a newline.

Compiling and Running

- Simplest: gcc hello.c
  Results in an executable called a.out
- But it’s nice to have a meaningful name
  gcc **–o hello** hello.c
- It’s also nice to take advantage of “recent” improvements
  gcc –o hello **–std=c99** hello.c
- And it would be nice to see more warnings
  gcc –o hello –std=c99 **–Wall** hello.c
- The compiler needs a little more help to let us know if we use uninitialized local variables, so here we turn on optimizations.
  gcc –o hello –std=c99 –Wall **–O1** hello.c
- Sometimes you need a -D option, e.g. **-D _XOPEN_SOURCE=500**, for the compiler to recognize certain symbols. These are given in the man pages.

Makefiles (minimal)

```
hello: hello.c
	gcc -o hello -std=c99 -Wall -O1 hello.c
```

- To avoid having to type that every time, we can create a “make file”, usually named Makefile.
- The first line

  hello: hello.c

  says that in order to create hello, we depend on hello.c
- If the file hello has a more recent timestamp than hello.c then the makefile won’t do anything.
- The second line says what to do if hello.c is more recent than hello.
- There must be a tab before the command
- And be sure to have a newline at the end of the last text line

Makefiles (all / clean)

```
# Makefile

all: hello

hello: hello.c
	gcc -o hello -std=c99 -Wall -O1 hello.c

clean:
	rm -f hello
```

- Comments are marked with a pound sign (#)
- The first item is the one done by default. Convention has us call it “all”
- Convention also has an option “clean”
  Note that there is no dependency

Makefiles (variables)

```
# Makefile
FLAGS = -std=c99 -Wall -O1

all: hello goodbye

hello:          hello.c
	gcc ${FLAGS} -o hello hello.c

goodbye:  goodbye.c
	gcc ${FLAGS} -o goodbye goodbye.c

clean:
	rm -f hello goodbye
```

Strings

- In C++ we have the string class.
  Obviously, in C we do not.
- C's strings are just arrays of characters that have
  a null character ('\0') at the end.
- C also has a collection of useful functions in string.h:
  - `size_t strlen(const char *s)`
  - `int strcmp(const char *s1, const char *s2)`
  - `char *strcpy(char *target, const char *source)`
  - `char *strncpy(char *target, const char *source, size_t n)`
  - `char *strcat(char *target, const char *source)`
  - `char *strncat(char *target, const char *source, size_t n)`
  - `char *strstr(const char *string, const char *substring)`

String to Integer

- How to convert a string to an int? What would you do in C++?
- ```
  int atoi(const char *string) // #include <stdlib.h>
  int val = atoi("1234")              // val == 1234
  int val = atoi("567xyz")            // val == 567
  int val = atoi("  89")              // val == 89
  ```
- But there’s a problem...
- What if string is not an int? What should atoi return?
- To check for errors, use:

  ```
  long int strtol(const char *nptr,
                  char **endptr, // where digits end
                  int base);
  ```
  ```

Byte Array Functions

- `strings.h`
  `void bzero(void *s, size_t n)`
- `string.h`
  `memset(void *b, int c, size_t n)`

C Standard I/O: Opening a file

```
#include <stdio.h>
FILE *fopen( const char *path, const char *mode )
```

Mode

- r: reading. Stream positioned at beginning.
- r+: read/write. Stream positioned at beginning.
- w: write. Create or truncate.
- w+: read/write. Create or truncate
- a: append. a+: append and read.

Created files have default permissions: 0666

S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH

Return value:

FILE* for the opened file, if successful.

NULL otherwise, with errno set to specify the error

C Standard I/O: Closing a file

```
int fclose(FILE *fp)
```

Return value: 0 for success, EOF for failure with errno set.

C Standard I/O: Line Input

- `char *fgets(char *buf, int count, FILE *fp)`
  buffer <u>will</u> include the new-line character but only <u>if</u> it fits.
- `char *gets(char *buf)`
  buffer <u>will not</u> include the new-line character.
  caller has to make sure that the buffer is large enough.
  Removed in C11 (but gcc hasn't done so yet)
- Some installations will provide a non-standard function:
  ```
  ssize_t getline(char **lineptr, size_t *n,
                  FILE *stream);
  ```
  Do not depend on it.

C Standard I/O: Line Ouput

- `int puts(const char *buf);`
  Writes to standard output, appending a newline character
- `int fputs(const char *buf, FILE *fp)`
  Writes to fp. Does not append a newline character.

C Standard I/O: Seeking

- `int fseek(FILE *stream, long offset, int whence)`
  - `whence:`
    - `SEEK_SET`
      `Offset` from the beginning. You are “setting” the the point to do the next I/O to offset
    - `SEEK_CUR`
      Offset from the current location.
    - `SEEK_END`
      Offset from the current end. Note that positive offsets extend beyond the end of the file.
  - Returns 0 for success and -1 otherwise.
- `long ftell(FILE *stream)   // Where are we?`

Standard Streams

- Standard input
  - By default, the keyboard
  - In C++, known as cin
  - In C, known as stdin
- Standard output
  - By default, the screen
  - In C++, known as cout
  - In C, known as stdout
- Standard error
  - By default, the screen
  - In C++, known as cerr
  - In C, known as stderr

Formatted Output

```
#include <stdio.h>

int main() {
    int x = 42;
    double pi = 3.141592653589;
    char cat[] = "felix";
    printf("x = %d, pi = %.10f, cat = %s\n", x, pi,
cat);
    return 0;
}
```

- `printf("Some control string", arg1, arg2, …)`
- Returns count of characters printed or -1.
- %d, %f, %x, %o, %p, %s, %c
- %10d - right justify in a field 10 wide. Pad with blanks

fprintf, s[n]printf

- fprintf allows you to print to a stream other than standard output:
```
fprintf(stderr, "This will go to standard error");
```
- sprintf allows you to print to a string.
```
char buf[80];
sprintf(buf, "pi = %f", 3.1415926535);
```
- What if the char array is not large enough?
- Overwriting of memory. Bad.
  (Ever hear of buffer overflow hacks?)
- Better to use snprintf (from C99)
```
snprintf(buf, 80, "pi = %f", 3.1415926535);
```

Character I/O

Input

- int fgetc(FILE* fp)
- int getc(FILE* fp)
- int getchar(void)

Output

- int fputc(int c, FILE* fp)
- int putc(int c, FILE* fp)
- int putchar(int c)

- getc and putc *may be* macros, evaluating fp more than once.
- getchar uses getc and putchar uses putc.
- All functions return the character read or written if successful.
  Or EOF on failure.

struct

```
struct MyStruct {
    int x;
    int y;
};

int main() {
    MyStruct mine;
    mine.x = 42;
}
```

- struct is familiar from C++, so what's the point here?
- Line 7: error: `MyStruct' undeclared (first use in this function)
- Huh?

Using struct Correctly

```
struct MyStruct {
    int x;
    int y;
};

int main() {
    struct MyStruct mine;
    mine.x = 42;
}
```

- Ok. The compiler now knows that MyStruct is a struct.
- Structs are in a different “namespace” than variables and functions.
- This allows you to have a variable and a struct with the same name. (Good idea?)
- It requires that you say that a type is a struct everywhere you use it.

Using a struct Without Having to Say

```
typedef struct MyStruct {
   int x;
   int y;
} YourStruct;

int main() {
   struct MyStruct mine;
   mine.x = 42;
   YourStruct yours;
   your.x = 17;
}
```

- C programmers and C libraries seem to prefer the more verbose approach
- Exceptions: FILE, DIR …
- You can also use:
  typedef struct { int x; int y; } YourStruct;

Dynamic Memory

- There isn’t any new / delete.
- Heap space is allocated with
  - `void *malloc(size_t size)`
    - Does not zero
  - `void *realloc(void *ptr, size_t newsize)`
    - Does not zero
    - Will literally extend the space if possible.
    - Will handle copying to a larger memory block if could not extend.
  - `void *calloc(size_t nobj, size_t size)`
    - Does zero. Note this is the only one that does.
- On failure, all return NULL.
- Generally implemented with system call sbrk.
- Note that in C, void * does not have to be explicitly cast when assigning to/from other pointer types
- Heap space is freed with
  - `void free(void *ptr)`
    - No return value! No need to check for errors!

Not Passing Parameters

```
void foo() {
    puts("Hello world");
}
void bar(void) {
    puts("Hello world");
}
int main() {
    foo();
    foo(17);    // Will compile!
    bar(17);    // Will not compile!
    return 0;
}
```

- What if your function is not expecting any arguments?
- Specify void in the parameter list!
- Otherwise C compiler *will* allow arguments to be passed!

Parameter Passing

- All parameter passing is by value.
- There is no pass by reference.
- That's easy to understand when passing int, char or double.
- What about pointers?
    - Note, some literature will use the phrase pass-by-reference when you are explicitly passing an address.

Pointer Swap

```
void swapPtr(int *a, int *b) {
   int *tmp = a;
   a = b;
   b = tmp;
}
void display(int x, int y, int *p, int *q) {
   printf("x = %d, y = %d\n", x, y);
   printf("*p = %d, *q = %d\n", *p, *q);
   printf("========================\n");
}
int main() {
   int x = 17, y = 42, *p = &x, *q = &y;
   printf("Original:\n");
   display(x, y, p, q);
   swapPtr(p, q);
   printf("After swapPtr:\n");
   display(x, y, p, q);
}
```

Pointer Swap

- What happened?
- Nothing.
- Why?
- The pointers were passed by value.
- Only the copies of the pointers were changed.
- How should we write it?

Pointer Swap

```
void swapPtr(int **a, int **b) {  // Passing addresses of pointers
    int **tmp = a;
    a = b;
    b = tmp;
}

void display(int x, int y, int *p, int *q);  // Nothing changed

int main() {
    int x = 17, y = 42, *p = &x, *q = &y;
    printf("Original:\n");
    display(x, y, p, q);
    swapPtr(&p, &q);  // Swapping pointer addresses, not contents
    printf("After swapPtr:\n");
    display(x, y, p, q);
}
```

Pointer Swap

- What happens now?
- Still nothing.
- What’s wrong this time?
- a and b in swapPtr are **still** copies.
  Exchanging them does not accomplish anything.
- We have swap what a and b point at!

Pointer Swap - Success!

```
void swapPtr(int **a, int **b) {    // Same parameters
   int *tmp = *a;  // Swapping what they point to!
   *a = *b;
   *b = tmp;
}

void display(int x, int y, int *p, int *q);  // Nothing changed

int main() {  // Nothing changed in main
   int x = 17, y = 42, *p = &x, *q = &y;
   printf("Original:\n");
   display(x, y, p, q);
   swapPtr(&p, &q);
   printf("After swapPtr:\n");
   display(x, y, p, q);
}
```

Size of an Array

```c
void foo(int arr[100]) {
    printf("sizeof(arr): %d\n", sizeof(arr));
}

int main(void) {
    int x[100];
    printf("sizeof(x): %d\n", sizeof(x));
    foo(x);
}
```

- What is the size of an array? What will this program display?
- main shows that x has a size of 400.
- What happens when we pass an array?
- Arrays cannot be automatically copied.
- Only the address of the array is passed.
- foo prints
- sizeof(arr): 4

Function Overloading

- In C++, we can give two different functions the same name, so long as their parameter lists are different.
  - void foo(int);
  - void foo(char*);
- In C, function overloading is not legal.
  We would have to do something like:
  - void fooInt(int);
  - void fooString(char*);
- Some people recommend using the C approach even in languages like C++ where it is not required.

But...

- The Unix api has two versions of the function “open”
- `open(pathname, open_flags)`
- `open(pathname, open_flags, permissions)`
- How can that work without function overloading?
- They use the same technique that printf uses...
- Variable-length Argument Lists

Variable Length Argument Lists

- Uses a new type, va_list, which acts as a “pointer” to the argument list.
- Macros:
  - va_start. Sets the va_list variable to “point to” the first argument in the variable-length portion of the list. (We actually pass it the name of the *last* fixed position parameter.)
  - va_arg. Allows the code to specify the type of the next argument in the list, returns it and bumps the va_list variable to the next item.
  - va_end. Cleans up when we are all done.
- The <u>program</u> (not the compiler) needs to know how many arguments to expect. This might be a value in one of the fixed parameters.
- There must be at least one fixed-position parameter.
- <stdarg.h>

Variable Length Arg Example

```
#include <stdarg.h>
#include <stdio.h>
void foo(int n, ...) { // Yes, really "..."
   va_list ap;
   va_start(ap, n);
   for (int i = 0; i < n; ++i) {  // Assuming the next n are strings
      char *s = va_arg(ap, char*);
      puts(s);
   }
   double dub = va_arg(ap, double); // And after the strings is a double
   printf("some double: %f\n", dub);
   va_end(ap);
}
int main() {
   foo(3, "moe", "larry", "curly", 3.14159);
}
```

Casting in C vs C++

- C++ casting:
    - static_cast<T>
        - Cast between “related” types
    - redefine_cast<T>
        - Cast between “unrelated” types, e.g. int and double*
    - const_cast<T>
        - Cast away constness
    - dynamic_cast<T>
        - Used with inheritance
- In C, to cast a value x to a type T: (T)x

C89 vs C99

- C89 requires local variables to be defined before any code.
    - C99 allows local variables to be declared wherever you like, similar to C++.
    - NB: gcc uses this C99 extension by default.
- C89 requires that main have a return statement,
  as with any other function.
    - C99 acts like C++, allowing main to leave off the return, returning 0 by default.
- C89 & C++ require local array sizes to be known at compile time.
- C99 allows arrays to be declared with a size determined at runtime.
    - We will <u>not</u> use this feature of C99.

Most Common Includes

In C, the most common includes you will want are:

- stddef.h: standard types and consts
  - size_t, NULL
- stdio.h: standard C i/o functions.
  - printf, scanf
- string.h: C string manipulation functions
  - strlen, strcpy, strcmp, strcat
- stdlib.h: other standard C functions
  - exit, rand, malloc, free, qsort.
- ctype.h: character manipulation
  - isdigit, tolower

Function Pointers

- How to pass a function?
- C++ programmers tend to use functors to “pass a function” but that mechanism requires classes.
- C passes functions using function pointers.
- But like all pointers, they need to have a type.
- How do you specify the type of a function?
- Do all functions have the same type?
- No

Function Pointer Example

- Some function’s definition:
  - `void doNothing(void) {}`
- Declaration for a function pointer to match:
  - `void (*f)(void);   // f is a function pointer`
- Use:

```
void takesFunction( void (*f)(void) ) {
   (*f)();  // official way
   f();     // also accepted
   *f();    // Oops
}
int main() {
   takesFunction(doNothing);
}
```

Implicit Declarations

```
// implicitDeclaration.c
int main() {
   printf("Hello\n");
}
```

- What happens when we compile?

  implicitDeclaration.c: In function 'main’:

  implicitDeclaration.c:3: warning: incompatible implicit declaration of built-in function 'printf’

- Incompatible? With what? Where is printf declared?
- C++ <u>requires</u> that a function be declared or defined before it is used.
- C <u>allows</u> the compiler to infer the declaration
- Use -Wall to generate the warning:
- implicitDeclaration.c:2: warning: implicit declaration of function 'printf'

Tokenizing

- How would you “tokenize” a line in C++?
    - istringstream
- In C, one way is strtok:

```
char arr[] = “   one  two three”;
char *s = strtok(arr, “ ”); // Get first token
while (s != NULL) {
   printf(“token: %s\n”, s);
   s = strtok(NULL, “ ”);  // Get next token
}
```

- But what's going on? How does this work? How is strtok returning something different each time in the loop when it's called with the same arguments?

static

- How did strtok know where you were in all those calls after the first?
- It has a “static” local variable.
- static variables remember their values between calls.
- Other examples
- [see printTime.c on next slide]

```c
/*
    printtime.c
*/
#include <stdio.h>  // printf
#include <time.h>   // time, time_t, ctime
#include <unistd.h> // sleep
int main() {
   time_t now = time(NULL);
   char* t1 = ctime(&now);
   printf("t1: %s", t1);
   sleep(5);
   now = time(NULL);
   char* t2 = ctime(&now);
   printf("t2: %s", t2);
   printf("t1: %s", t1);
}
```

Macros

- `#define FRED`
- `#define FRED 1`
- `#define RIGHT (i + 1) % N`
- `#define square(x) x * x`
  - `square(5)`
    - `5 * 5`
  - `square(1+1)`
    - `1 + 1 * 1 + 1  // Oops.  Result is 3.`
- Fix with: `#define square(x) ((x) * (x))`
  - But: `square(++a);`
    - `((++a) * (++a))`

Command Line Arguments

- Not actually different from C++, but you will use it more in our programs.
- `int main( int argc, char *argv[] ) { … }`
    - argc is the number of arguments
    - argv is an array of the arguments
        - The first entry in argv is the name of the executable being run. argc is at least 1.
- What if an argument is an integer?
  We still get it as a C string.

Environment Variables

- Every process has a set of “environment” variables, set up by the process’ s creator.
- What’s the point? Allows you to set up “defaults” for the way a program runs
- You can see them with the command: env
    - env [-i] [name=value] … [utility [arg …]]
        - -i says to ignore inherited environment completely
          Otherwise only replace specified names
- If no utility program is provided in the command,
  displays the resulting environment
- Where are these provided? Up above the call stack.

Display Environment

```
#include <stdio.h>

extern char **environ;

int main() {
    for (int index = 0; environ[index] != NULL; ++index) {
        puts(environ[index]);
    }
    for (char **p = environ; *p != NULL; ++p) {
        puts(*p);
    }
}
```

errno

- Many library and system calls return -1 or NULL
  if there is an error.
- But what was the error?
- For that you check the global int value errno.
- Possible errno values for printf include:
  EINTR: A signal interrupts the write before it could be completed.
  EIO: An I/O error occurs while writing
- For a readable error message, you can call `perror(const char *msg)`
  - Prints your message, if any, followed by a description of the error.

Done

whew!