

CME 212: C programming I

Lecture 3

January 18, 2012

C Programming Language

- Developed at Bell Labs by Dennis Ritchie in 1972
 - Used for the UNIX OS
 - Successor to “Algol” and “B”
- Minimalistic language
 - Simple parsing and compilation
 - Simple mapping to machine code
 - Support direct access to memory
 - Portable

Variants

- Kernighan and Ritchie (K&R)
 - Informal description by the “The C Programming Language” book
- Rigorous standards
 - ANSI X3.159-1989 (ANSI C, C89)
 - ISO/IEC 9899:1990 (C90)
 - ISO/IEC 9899:1999 (C99)
 - ISO/IEC 9899:2011 (C11) just ratified
- Compiler extensions
 - gnu89, gnu99, gnu11 (coming soon)

One very practical difference is C++ style line comments (//)

Language and Library

- The standards define the language and a standard library containing useful extensions
 - I/O, sorting, conversions, math, ...
- The language itself is very small
- If compiled using a standards compliant compiler, and using only the standard library, the code is portable

Syntax

- Source is free form
- Lines delimited using the semi-colon “;” character
- Source code contains
 - Declarations
 - Variables
 - Function prototypes (declarations)
 - Function definitions
 - Statements

Source Code

- Source code can be split up into several files
- Put together by the C preprocessor (cpp)
 - `#include`
- Typical suffix is “.c”
- After preprocessing the code is passed to the actual compiler

Variable Declarations

- A variable is declared using a type and a variable name, for example `int i = 1;`
- The type specification sets the representation and available operations

Table: Basic C types

type	representation
<code>int</code>	integer
<code>float</code>	real number
<code>double</code>	real number
<code>char</code>	ASCII character, byte

- There are also a few type qualifiers: `short`, `long`, `signed`, `unsigned`

Examples

Multiple variables of the same type can be defined in the same statement if they are separated by a comma “,”.

```
1| int a, b, c, d;
```

You can do assignments at the time of declaration.

```
1| double mmmm_pi = 3.14159;
```

Characters go between single quotes.

```
1| char q = 'z';
```

Be careful with pointers

```
1 | double* a, b;
```

Is b a pointer?

Available variable and function names

- Composed of letters, digits, underscores
- Must begin with a letter or underscore
 - underscores at the beginning or end of a variable often have some meaning in a project
- Reserved names (C keywords):

auto break case char complex const continue default do double else
enum extern float for goto if imaginary inline int long register
restrict return short signed sizeof static struct switch typedef union
unsigned void volatile while

Basic Integer Types in C (C++)

- **Characters:** `char`, `signed char`, `unsigned char`
(smallest chunk of memory we can access in C)
- **Short integers:** `short`, `short int`, `signed short`,
`signed short int`, `unsigned short`, `unsigned short int`
- **Integers:** `int`, `signed int`, `unsigned int`
- **Long integers:** `long`, `long int`, `signed long`, `signed long int`,
`unsigned long`, `unsigned long int`
- **Long long integers:** `long long`, `long long int`,
`signed long long`, `signed long long int`,
`unsigned long long`, `unsigned long long int`

Guaranteed Limits of ANSI C

Table: Standard C limits

type	guaranteed range	note
signed char	127 to 127	
unsigned char	0 to 255	
char	depends on system	
short int	32,768 to 32,767	
unsigned short int	0 to 65,535	
int	$\pm (2^{31}-1)$	
unsigned int	0 to $2^{32}-1$	
long int	$\pm (2^{31}-1)$	maybe 64-bit
unsigned long int	0 to $2^{32}-1$	maybe 64-bit
long long int	$\pm (2^{63}-1)$	always 64-bit
unsigned long long int	0 to $2^{64}-1$	always 64-bit

Note: the standard ranges allow for ones-complement negation.
`int`'s are signed by default. You can often omit the `int` keyword.

For reference

$2^{31}-1$	2,147,483,647	\approx 2.1 billion
$2^{32}-1$	4,294,967,295	\approx 4.3 billion
$2^{63}-1$	9,223,372,036,854,775,807	\approx 9.2 quintillion
$2^{64}-1$	18,446,744,073,709,551,615	\approx 18.4 quintillion

Types for floating-point

- There are three types
(`float` < `double` < `long double`)
- Precision is dependent on which floating point model is chosen.
- Typical exponents are 10^{38} , 10^{308} , 10^{4932} respectively
- Some models have also implemented runtime exceptions for not-a-number (NaN), overflow, underflow, division by zero
- Compilers often give you a choice of floating-point models. Some are faster than others (speed vs. accuracy)
- IEEE 754 is the widely used standard, allows repeatable computations across machines (sort of)

Function prototypes

Function prototypes or declarations define the interface to a function:

- Return type of function
- Name of function
- Number of arguments
- Types of arguments

Examples:

```
1 | float power(float base, float);  
2 | void set_grade(char grade);
```

Notes: `void` indicates no type, variable names need not be specified in prototypes.

Function definitions

- Contain declarations and statements
- Function body is delimited by “curly brackets” { }
- Returns using return keyword

Examples:

```
1 float power(float base, float exponent)
2 {
3     float result;
4     result = ... ;
5     return result;
6 }
```

Your first C program

```
1 int main(void)
2 {
3     /* This is a comment */
4     int a; /* This is a variable declaration */
5
6     // This is C99 comment
7     a = 5; // This is a statement
8
9     return 0;
10 }
11 /* these comments
12    can span
13    lines
14 */
```

Header files

- In many cases it is wise to separate the function prototype from the definition
- Typically, function prototypes are stored in a so called **header file** (.h)
- The actual code for the functions are then stored in another file, a **source file** (.c)
- This file can be compiled into an **object file** (.o)
- **Header files** should have a “header guard” to prevent multiple includes

Preprocessor Includes

- If you want to call a function from sources files
- You include the header file using the preprocessor
 - `#include "my_file.h"`
- This tells the compiler that there is a function called so-and-so and that code for that function will be provided at link time
 - Defines a contract, you promise to provide code for the prototypes
- A **source file** should always include its **header file**

C Expressions

You can think of expressions in C as phrases in English.

- An **object** in C lingo is a region of memory that can be inspected and modified
- An **lvalue** is an expression that refers to an object
- Only **lvalues** can be on the left-hand side of an assignment

Examples:

```
1 | int A;  
2 | A = 5; // A is an lvalue  
3 | if ( A < 6 )  
4 |     printf("it's party time!\n");
```

Important differences compared to Java and C++

- All declarations of variables must be made before any statements follow (not true in C99)
- Comments are written using the `*.*` pair. (`//` is allowed in C99)
- No exception handling (try-catch blocks)
- Many other features: classes, templates... gnu c ma

Scope

Scope refers to the visibility of variables in the context of a running program.

- Variables declared inside a function (including `main()`) are visible only to that function
- Variables declared outside any function (global) are visible only to that file
- To increase the scope to other files you may use the `extern` modifier (binding is deferred to linking)

file1.c

```
1 // global scope
2 int errcnt = 0; // defines the storage
```

file2.c

```
1 // global scope
2 extern int errcnt; // only declares the name
```

Extent

- Variables and functions have existence at runtime (associated storage)
- Static extent
 - Storage is provided during entire program lifetime
 - Functions, variables with file or external scope have static extent
- Local extent
 - Storage is allocated during function call
 - Parameters, local variables
 - Referred to as “automatic” variables (they automatically appear and go away)

Example

```
1 // static extent
2 int a = 5;
3 static int dummy;
4
5 int f(int);
6 static int g(int);
7
8 int main(void) {
9     int b; // local extent, automatic
10    b = f(a);
11 }
12
13 int f(int input) {
14     int d; // local extent, automatic
15     static int c; // static extent
16
17     dummy = input;
18     d = g(a);
19     return d;
20 }
```

- Static extent can be toggled with the `static` keyword
- Variables and functions declared with `static` are not exported to the linker, allows for variables and functions that are private to the file

Initializations

- Static variables are initialized to zero
- Values of automatic variables are undefined unless initialized explicitly
 - `int a; // a is not initialized`
 - `int b = 1; // b is initialized`
- A variable declaration with local extent makes sure that there is storage associated with the name
 - Content is undefined

Enums

Enumerations or enums allow for designation of sets. They are represented as integers, but often make code easier to read.

```
1 enum days { mon, tue, wed, thu, fri, sat, sun };
2
3 enum days today = mon;
4
5 if ( today == mon )
6     printf("let's have a party!\n");
```

Arrays

Arrays are used to store a sequence of data of the same type.

```
1 int a[42];
2 int b[3] = {1,2,4};
3 int n = 10, d;
4 int mat[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
5 // C uses 0-based indexing
6 int d = a[0];
7
8 int f(int n) {
9     // a variable length array VLA, allowed in C99
10    // illegal for static or extern
11    int elements[n];
12 }
```

Strings

- Strings in C are arrays of char
- Terminated by the '\0' null character

```
1 char first_name[4] = {'B', 'o', 'b', '\0'};  
2 char last_name[] = "Jones";  
3 // last_name is a character array of length 6  
4 // last_name[5] == '\0'
```

- Forgetting the termination character is a bad mistake!

Structures

C structures allow the aggregation of multiple data fields into a single entity.

```
1 struct point { double a,b; };
2
3 // structure can be nested
4 struct triangle {
5     struct point a,b;
6     struct point c;
7 };
8
9 // we can initialize at definition
10 struct triangle t = {{0.1,0.2},{1.0,2.0}};
11
12 // access data using .fieldname
13 t.a.a = 3.0;
```

Pointers

- Pointers are variables whose values are references to other variables
 - In C they are defined as addresses
- Declared using *

```
1 int a,b;
2 int *p,*q; // pointer declaration
3
4 a = 5;
5 b = 6;
6
7 // & is the "address of" operator
8 p = &a; // p now stores address of a
9 q = &b; // q now stores address of b
10 p = q; // p now stores the address of b
```

Type Casting

- You can override the type system using type casting
- Can be very dangerous
 - You must know what you are doing
 - Different types have different representations

```
1 int a;  
2 unsigned long int b;  
3 float c;  
4  
5 a = 5;  
6 b = (unsigned long int) a;  
7 c = (float) b;
```

Automatic conversions

Binary operators like `+`, `-`, `*`, `/` convert “lower” or smaller types to “higher” or larger types:

- 1 If either operand is `long double`, convert the other to `long double`
- 2 Otherwise, if either operand is `double`, convert the other to `double`
- 3 Otherwise, if either operand is `float`, convert the other to `float`
- 4 Otherwise, convert `char` and `short` to `int`
- 5 Then if either operand is `long`, convert the other to `long`

Things get strange with unsigned integer types!

Generic Pointers

Because pointers are addresses, we can use type casting and a special void pointer to construct a generic pointer

```
1 | int a = 5;  
2 | float b = 8;  
3 | void *c;  
4 |  
5 | c = (void *)&a;  
6 | c = (void *)&b;  
7 | c = (void *)0x234334;
```

Calculating the Size of Variables

- C has a built in function `sizeof` that gives the size in bytes of the representation used
 - Non-composite variables typically require 1,2,4 or 8 bytes
- `sizeof` on structs gives the size of the entire composite type, i.e. the aggregate size of all the members
- `sizeof` on arrays gives the size of the entire array, i.e.
 - `sizeof(element_type)*number_of_elements`

Pointer Arithmetic

- The values of pointers are addresses
- Adding one to a pointer does not increment the address by one
- It increments the address by the size of the type, given by the function `sizeof()`
- Gives the correct addresses for indexing an array

```
1 | int a[5] = {1,2,3,4,5};  
2 | int *p = a;  
3 | for (int i=0; i !=5; ++i)  
4 |     printf("%d\n", *(p+i));  
5 | // will print 1 2 3 4 5 on separate lines
```

Arguments to Functions

- All arguments are passed call-by-value
 - Local dummy variables are created and the input values copied to these
- All elements of structs are copied
- Does not apply to arrays
- Arrays are type-casted into pointers, and pointer value (memory address) is passed

Call-by-reference

- Call-by-value creates some overhead as the arguments need to be copied
 - Large structs
 - Many arguments
- By passing a pointer we can reference the object using this reference
- The value of the pointer variable (an address) is passed call-by-value (copied)

Many Output Arguments

- Use a regular return argument and pack your arguments in a struct
 - However, this causes lots of copying
- Use pointers!

```
1 // returns a struct
2 struct my_args f(struct my_args);
3
4 // pass pointers
5 // f can now change the value at the address
6 void f(int *arg1, float *arg2, double *arg3);
```

Arguments to Functions, Example

```
1 struct big_one {
2     int a,b,c,d,e,f,g,h,I,j,k,l,m,n,o;
3 };
4 int f(struct big_one *b, int array[]);
5 int main(void) {
6     struct big_one a_big_one;
7     int array[10];
8
9     a_big_one.d = 5;
10    (void)f(&a_big_one,array);
11    return 0;
12 }
13
14 int f(struct big_one *b, int array[]) {
15     b->d = 6; // b->d is equivalent to (*b).d
16     array[b->a] = 23;
17     return b->k;
18 }
```

Returning Pointers

```
1 char *what_do_I_do(char *s) {  
2     while( *s != '_' && *s != '\0' ) s++;  
3     if ( *s == '\0' )  
4         return NULL;  
5     else  
6         return s;  
7 }
```

Never return the address of automatic variables since they die when the function returns

```
1 int *dont_do_this() {  
2     int a;  
3     return &a;  
4 }
```