Programming in C

UVic SEng 265

Daniel M. German
Department of Computer Science
University of Victoria

October 15, 2002 Version: 1.00

C

- Developed by Brian Kernighan and Dennis Ritchie of Bell Labs
- Earlier, in 1969, Ritchie and Thompson developed the Unix operating system
- We will be focusing on a version referred to as ANSI/ISO C.

What is C?

- ”General Purpose” programming language
- Simple language to compile
- Simple constructions, very close to the hardware level
- Low overhead of execution
  - no run-time checking
  - no array access bounds-checks
  - no null-pointer checks
  - no checks on uninitialized variables

What is C? ...

- Very terse programming language
- Keywords are often optional
- Limited native functionality; no built-in libraries or utilities beyond the C library (no GUI, no database, no complex datatypes, mathematical functions, etc.)
- Nowadays almost any architecture has a C compiler

Hello, World!

- hello.c

```
#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");
    return 0;
}

% gcc -pedantic -Wall -ansi -c hello.c # produces hello.o
% gcc -o hello hello.c                # produces hello
% ./hello
Hello, World!
```


Basic syntax

- similar to Java (Java is a descendant of C++, which is descended from C)

	C	Java
comments	/* ... */	//, /* ... */
basic types	char (byte)	char (16-bit Unicode)
	int (natural)	byte (1-byte signed)
	short int (smaller or equal to int)	int (4)
	long int (bigger or equal to int)	short (2)
	float (4 byte IEEE)	long (8)
	double (8 byte IEEE)	float (4 byte IEEE)
		double (8 byte IEEE)
aggregate types	array	Vector
	struct	class

Important details

- Ground rules:
  - All C programs must have a function called `main`
  - keywords are always lowercase
  - statements must be terminated with a semicolon
- Defining variables
  - general syntax
    `type name;`
  - declaration with initialization:
    `type name = value;`

Programming style

- Any amount of white space is a considered a single space
- End of lines, tabs and spaces can be liberally used
- More white space can help improve code readability
- Commenting is extremely important for maintaining code
- Use indentation in conjunction with curly braces ({, }) to indicate different levels of nested functions.
- K&R is the style for this course:
  - Opening brace of function in column 1
  - Opening brace in the same line as expression (if, while, for)
  - The block inside the braces should be indented
  - End brace at same level of opening statement

Program Style...

- Indentation = 4 spaces
- No lines beyond 80 characters!

```
#include <stdio.h>

int get_line(char inputString[], int maxSize)
{
    int c, i;

    for (i=0; i<maxSize-1 &&
            (c=getchar())!=EOF && c=='\n'; ++i)
        inputString[i] = c;

    if (c == '\n') {
        inputString[i] = c;
        ++i;
    }

    inputString[i] = '\0';
    return i;
}
```


Type declarations

- Examples:

```
int y;        /* declaration of integer y */
int x = 4;    /* declaration with init. */
int a, b, c;  /* decls. of ints a, b and c */
int a, b = 1; /* decl of a, and b, initialize b */
char c = 'A'; /* not the same as 'a' */
```


Signed vs. unsigned integers

- On *aserver*, a 4-byte signed integer may represent the values from -2,147,483,648 (`MIN_INT`) to 2,147,483,647 (`MAX_INT`)
- Signed numbers must reserve one bit to represent sign (complement 2 representation)
- 4-bytes == 32 bits (8-bit bytes); with 1 bit left over, maximum value is $2^{31} - 1$; minimum value is -2^{31}
- An unsigned 4-byte integer can only be positive and has a range from 0 to 4,294,967,295 (`MAX_UINT`)
- By default a variable is signed
- Unsigned declarations:

```
unsigned char c; unsigned int i; unsigned long l;
```


Arrays

- Elements are indexed, starting from 0 and going up to (size-1)

```
int num[10];
num[0] = 4510;
num[1] = -3312;
num[9] = 214;
num[10] = 31441;              /* non existent */
```

- format for one-dimensional array declarations:

```
<type> <var name>[<size>]
```

- Example:

```
float   vector[3];
char    buffer[256];
```

- `<size>` must be known at compile time

Control flow

- Four basic constructs:
  - if-then, if-then-else (branching)
  - switch (multi-way branching)
  - while loops, do-while loops
  - for loops
- meaning of these constructs is very similar to that in Java
- however, major difference is in the meaning of branch- and loop-conditions
- we will not use `goto` or `setjmp/longjmp` in this course (but they are useful, so don't hate them!)

if statement

```
if (condition) {
    stmt1;
    stmt2;
} else if (condition2) {
    stmt3;
    stmt4;
} else {
    stmt5;
}
```


Multi-way branch (switch)

```
switch (condition) {
case value1:
    stmt1;
    break;
case value2:
    stmt2;
    break;
default:
    stmt3;
    break;
}
```


While loops

- Two varieties.
- Condition before the loop

```
while (condition) {
    stmt1;
    stmt2;
}
```

- Condition after the loop

```
do {
    stmt1;
    stmt2;
} while (condition);
```

for loops

```
for (expr1; condition; expr2) {
    stmt1;
    stmt2;
}
```

This is equivalent to:

```
expr1;
while (condition) {
     stmt1;
     stmt2;
     expr2;
}
```


Conditions

- C does not have a `boolean` type
- However, there are boolean like operators
- `&&, ||, ==, !=, !, >, <, >=, <=` (same as Java)
- Any expression that evaluates to non-zero is considered true

```
int x = 5;
if (x) {
    /* trivially true */
}
```


Careful!

- The assignment operator (=) and equality comparison operator (==) have different meanings
- Be careful not to confuse the two
- Legal (but possibly undesired) C code:

```
int a = 20;
if (a = 5) {   /* This expression is ALWAYS true */
    stmt1;
}
/* It should be a == 5  */
```


Functions

- general syntax:

```
<return type> name (<parameters>)
{
    <statements>
}
```

- parameter syntax:

```
<type> varname [, <type> varname>]
```

Functions ...

- example:

```
int main(int argc, char *argv[])
{
    printf("Hello, world!\n");
    return 0;
}
```

- example:

```
float max(float x, float y)
{
    if (x > y)
        return x;
    else
        return y;
}
```


Command line arguments

- passed into the `main()` function as parameters
- example: read a person's name from the command line
  `argc` number of parameters + 1
  `argv` contains executable name and each parameter

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    char *name;
    if (argc >= 2) {
        name = argv[1];
    } else {
        name = "anonymous";
    }
    printf("Hello %s!\n", name);
    return(0);
}
```


Command line arguments

- `argc` is the count of `argv` elements (which includes the name of the executable)
- `argc` must at least be 1
- `argv` is declared as an array of unknown length (note the absence of a number within the square brackets)
- `argv[]` are the command-line parameters
- `char *` is a pointer to some characters. More about pointers later.

Scope of a Variable

- Variables have a scope (area in which they are visible/available)
- In general: visible within the block in which they are defined

```
int Compute(int a)
{
    int x = 5;
    int y = 5;
    if (a == y) {
        int z;
        z = y + 3;
    }
    x = z;    /* compile-time error */
    return x;
}
```

Scope of a Variable ...

- Global variables, by default, are only available for only the file in which they are defined

```
int count = 0;

int main(void)
{
    count = 5;
    printf("%i", count);
}
```


Scope of a Variable...

- If a variable is used in a file, but is defined in another file, you must declare it before using it
- use `extern`
- example:

```
extern int count;

void increment(void)
{
    count++;
}
```


Function prototypes

- The code of a function is its **definition**
- A **prototype** is a **declaration**
- A declaration tells the compiler the type of a function, and the type and number of parameters
- A function should be **declared** before it is used
- If a function name is used, but it has not been declared, the compiler assumes its type and types of its parameters
- Many problems arise if this assumption is wrong

Function prototypes (contd)

- The declaration is known as the **prototype** of a function
- general syntax: `<return type> name(<parameters>);`
- parameters: types are necessary, but names are optional; names are recommended (improves code readability)
- examples:

```
int sort(int a[]);
float max3(float, float, float);
void error_message(char *m);
```

Function prototypes ...

- Prototypes are declarations
- In general, same as a normal function except there is no function body (i.e., absence of curly brackets) and is immediately terminated with a semi-colon
- Prototypes come at the top of file before any function is formally declared
- `#include <stdio.h>` is used to refer to all function prototypes for standard i/o (i.e., for `printf`, `fprintf`, `fread`, etc.)
- Prototypes help the compiler find programmer errors

Structures

- In some languages, referred to as ”records”
- Multiple variables inside a single structure (an ”aggregate”)
- Structure itself becomes a data type
- Can be thought of as an ancestor to objects

```
struct date {
    int month;
    int day;
    int year;
};
```


Using structures

- Once a structure variable is declared, variables inside the structure can be accessed using `<variable>.<field>` syntax:

```
struct date today;
today.day = 31;
today.month = 5;
today.year = 2001
```

- arrays of `struct`’s can also be defined

Other C data types

- Enumerations:
  - A ”translation table” can be established between integers and words
  - Using `enum`:

```
enum d_of_week { sun, mon, tue, wed, thur, fri, sat };
enum d_of_week today = mon;  /* today = 1 */
```

  - Note: the elements of enumerated data type are not strings, they are integers

Other data types ...

- Unions
  - Very similar to structs, but all its members share same memory
  - Each field becomes an “alias” to the same memory location

```
union my_union {
    char c;
    int i;
};
union my_union u;

u.c = 'A';
u.i = 80;
printf("Hello world [%d][%c]\n", u.i, u.c);

Hello world [80][P]
```