

Programming in C and C++

1. Types — Variables — Expressions & Statements

Dr. Anil Madhavapeddy

University of Cambridge
(based on previous years –
thanks to Alan Mycroft, Alastair Beresford and Andrew Moore)

Michaelmas Term 2014–2015

Structure of this course

Programming in C:

- ▶ types, variables, expressions & statements
- ▶ functions, compilation, pre-processor
- ▶ pointers, structures
- ▶ extended examples, trick hints 'n' tips

Programming in C++:

- ▶ references, overloading, namespaces, C/C++ interaction
- ▶ operator overloading, streams, inheritance
- ▶ exceptions and templates
- ▶ standard template library

Java native interface (JNI)

Text books

There are literally hundreds of books written about C and C++; five you might find useful include:

- ▶ Eckel, B. (2000). Thinking in C++, Volume 1: Introduction to Standard C++ (2nd edition). Prentice-Hall.
(<http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html>)
- ▶ Kernighan, B.W. & Ritchie, D.M. (1988). The C programming language (2nd edition). Prentice-Hall.
- ▶ Stroustrup, B. (2000). The C++ Programming Language Special Edition (3rd edition). Addison Wesley Longman
- ▶ Stroustrup, B. (1994). The design and evolution of C++. Addison-Wesley.
- ▶ Lippman, S.B. (1996). Inside the C++ object model. Addison-Wesley.

Past Exam Questions

- ▶ 1993 Paper 5 Question 5 1993 Paper 6 Question 5
- ▶ 1994 Paper 5 Question 5 1994 Paper 6 Question 5
- ▶ 1995 Paper 5 Question 5 1995 Paper 6 Question 5
- ▶ 1996 Paper 5 Question 5 (except part (f) `setjmp`)
- ▶ 1996 Paper 6 Question 5
- ▶ 1997 Paper 5 Question 5 1997 Paper 6 Question 5
- ▶ 1998 Paper 6 Question 6 *
- ▶ 1999 Paper 5 Question 5 * (first two sections only)
- ▶ 2000 Paper 5 Question 5 *
- ▶ 2006 Paper 3 Question 4 *
- ▶ 2007 Paper 3 Question 4 2007 Paper 11 Question 3
- ▶ 2008 Paper 3 Question 3 2008 Paper 10 Question 4
- ▶ 2009 Paper 3 Question 1
- ▶ 2010 Paper 3 Question 6
- ▶ 2011 Paper 3 Question 3

* denotes CPL questions relevant to this course.

Context: from BCPL to Java

- ▶ 1966 Martin Richards developed BCPL
- ▶ 1969 Ken Thompson designed B
- ▶ 1972 Dennis Ritchie's C
- ▶ 1979 Bjarne Stroustrup created C with Classes
- ▶ 1983 C with Classes becomes C++
- ▶ 1989 Original C90 ANSI C standard (ISO adoption 1990)
- ▶ 1990 James Gosling started Java (initially called Oak)
- ▶ 1998 ISO C++ standard
- ▶ 1999 C99 standard (ISO adoption 1999, ANSI, 2000)
- ▶ 2011 C++11 ISO standard (a.k.a. C++0x)

C is a “low-level” language

- ▶ C uses low-level features: characters, numbers & addresses
- ▶ Operators work on these fundamental types
- ▶ No C operators work on “composite types”
e.g. strings, arrays, sets
- ▶ Only static definition and stack-based local variables
heap-based storage is implemented as a library
- ▶ There are no `read` and `write` primitives
instead, these are implemented by library routines
- ▶ There is only a single control-flow
no threads, synchronisation or coroutines
- ▶ C seen as “a high-level assembly language” (take care!)

Classic first example

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, world\n");
6     return 0;
7 }
```

Compile with:

```
$ cc example1.c
```

Execute program with:

```
$ ./a.out
```

```
Hello, world
```

```
$
```

Produce assembly code:

```
$ cc -S example1.c
```

Basic types

- ▶ C has a small and limited set of basic types:

type	description (size)
<code>char</code>	characters (≥ 8 bits)
<code>int</code>	integer values (≥ 16 bits, commonly one word)
<code>float</code>	single-precision floating point number
<code>double</code>	double-precision floating point number

- ▶ Precise size of types is architecture dependent
- ▶ Various type operators for altering type meaning, including: `unsigned`, `long`, `short`, `const`, `volatile`
- ▶ This means we can have types such as `long int` and `unsigned char`
- ▶ C99 added fixed width types `int16_t`, `uint64_t` etc. as typedefs

Constants

- ▶ Numeric constants can be written in a number of ways:

type	style	example
<code>char</code>	<u>none</u>	<u>none</u>
<code>int</code>	number, character or escape seq.	<code>12 'A' '\n' '\007'</code>
<code>long int</code>	number w/suffix <code>l</code> or <code>L</code>	<code>1234L</code>
<code>float</code>	number with '.', 'e' or 'E' and suffix <code>f</code> or <code>F</code>	<code>1.234e3F</code> or <code>1234.0f</code>
<code>double</code>	number with '.', 'e' or 'E'	<code>1.234e3</code> <code>1234.0</code>
<code>long double</code>	number '.', 'e' or 'E' and suffix <code>l</code> or <code>L</code>	<code>1.234E3l</code> or <code>1234.0L</code>

- ▶ Numbers can be expressed in octal by prefixing with a '0' and hexadecimal with '0x'; for example: `52=064=0x34`

Defining constant values

- ▶ An enumeration can be used to specify a set of constants; e.g.:
`enum boolean {FALSE, TRUE};`
- ▶ By default enumerations allocate successive integer values from zero
- ▶ It is possible to assign values to constants; for example:
`enum months {JAN=1,FEB,MAR}`
`enum boolean {F,T,FALSE=0,TRUE,N=0,Y}`
- ▶ Names for constants in different `enums` must be distinct; values in the same `enum` need not
- ▶ The preprocessor can also be used (more on this later)

Variables

- ▶ Variables must be declared before use
- ▶ Variables must be defined (i.e. storage set aside) exactly once. (A definition counts as a declaration).
- ▶ A variable name can be composed of letters, digits and underscore (`_`); a name must begin with a letter or underscore
- ▶ Variables are defined by prefixing a name with a type, and can optionally be initialised; for example: `long int i = 28L;`
- ▶ Multiple variables of the same basic type can be declared or defined together; for example: `char c,d,e;`

Operators

- ▶ All operators (including assignment) return a result
- ▶ Most operators are similar to those found in Java:

type	operators
arithmetic	+ - * / ++ -- %
logic	== != > >= < <= && !
bitwise	& << >> ^ ~
assignment	= += -= *= /= %= <<= >>= &= = ^=
other	sizeof

Type conversion

- ▶ Automatic type conversion may occur when two operands to a binary operator are of a different type
- ▶ Generally, conversion “widens” a variable (e.g. `short` → `int`)
- ▶ However “narrowing” is possible and may not generate a compiler warning; for example:

```
1 int i = 1234;  
2 char c;  
3 c = i+1; /* i overflows c */
```

- ▶ Type conversion can be forced by using a cast, which is written as: (type) exp; for example: `c = (char) 1234L;`

Expressions and statements

- ▶ An expression is created when one or more operators are combined; for example `x *= y % z`
- ▶ Every expression (even assignment) has a type and a result
- ▶ Operator precedence provides an unambiguous interpretation for every expression
- ▶ An expression (e.g. `x=0`) becomes a statement when followed by a semicolon (i.e. `x=0;`)
- ▶ Several expressions can be separated using a comma ‘,’; expressions are then evaluated left to right; for example: `x=0,y=1.0`
- ▶ The type and value of a comma-separated expression is the type and value of the result of the right-most expression

Blocks or compound statements

- ▶ A block or compound statement is formed when multiple statements are surrounded with braces (`{ }`)
- ▶ A block of statements is then equivalent to a single statement
- ▶ In ANSI/ISO C90, variables can only be declared or defined at the start of a block (this restriction was lifted in ANSI/ISO C99)
- ▶ Blocks are typically associated with a function definition or a control flow statement, but can be used anywhere

Variable scope

- ▶ Variables can be defined outside any function, in which case they:
 - ▶ are often called global or static variables
 - ▶ have global scope and can be used anywhere in the program
 - ▶ consume storage for the entire run-time of the program
 - ▶ are initialised to zero by default
- ▶ Variables defined within a block (e.g. function):
 - ▶ are often called local or auto variables (register encourages the compiler to use a register rather than stack)
 - ▶ can only be accessed from definition until the end of the block
 - ▶ are only allocated storage for the duration of block execution
 - ▶ are only initialised if given a value; otherwise their value is undefined

Variable definition versus declaration

- ▶ A variable can be declared but not defined using the `extern` keyword; for example `extern int a;`
- ▶ The declaration tells the compiler that storage has been allocated elsewhere (usually in another source file)
- ▶ If a variable is declared and used in a program, but not defined, this will result in a link error (more on this later – and in the Compiler Construction course)

Scope and type example

```
1 #include <stdio.h>
2
3 int a;                               /*what value does a have? */
4 unsigned char b = 'A';
5 extern int alpha;                     /* safe to use this?      */
6
7 int main(void) {
8     extern unsigned char b;          /* is this needed?      */
9     double a = 3.4;
10    {
11        extern a;                     /*why is this sloppy?   */
12        printf("%d %d\n",b,a+1);     /*what will this print? */
13    }
14
15    return 0;
16 }
```

Arrays and strings

- ▶ One or more items of the same type can be grouped into an array; for example: `long int i[10];`
- ▶ The compiler will allocate a contiguous block of memory for the relevant number of values
- ▶ Array items are indexed from zero, and there is no bounds checking
- ▶ Strings in C are typically represented as an array of `chars`, terminated with a special character `'\0'`
- ▶ There is language support for this representation of string constants using the `"` character; for example:
`char str[]="two strs mer" "ged and terminated"`
(note the implicit compile-time concatenation)
- ▶ String support is available in the `string.h` library

Control flow

- ▶ Control flow is similar to Java:
 - ▶ `exp ? exp : exp`
 - ▶ `if (exp) stmt1 else stmt2`
 - ▶ `switch(exp) {
 case exp1:
 stmt1
 ...
 default:
 stmtn+1
}`
 - ▶ `while (exp) stmt`
 - ▶ `for (exp1; exp2; exp3) stmt`
 - ▶ `do stmt while (exp);`
- ▶ The jump statements `break` and `continue` also exist

Control flow and string example

```
1 #include <stdio.h>
2 #include <string.h>
3
4 char s[]="University of Cambridge Computer Laboratory";
5
6 int main(void) {
7
8     char c;
9     int i, j;
10    for (i=0,j=strlen(s)-1;i<j;i++,j--) /* strlen(s)-1 ? */
11        c=s[i], s[i]=s[j], s[j]=c;
12
13    printf("%s\n",s);
14    return 0;
15 }
```

Goto (considered harmful)

- ▶ The `goto` statement is never required
- ▶ It often results in code which is hard to understand and maintain
- ▶ Exception handling (where you wish to exit or `break` from two or more loops) may be one case where a `goto` is justified:

```
1 for (...) {
2   for (...) {
3     ...
4     if (critical_problem)
5       goto error;
6   }
7 }
8 ...
9 error:
```

fix problem, or abort

Exercises

1. What is the difference between 'a' and "a"?
2. Will `char i,j; for(i=0; i<10,j<5; i++,j++) ;` terminate? If so, under what circumstances?
3. Write an implementation of bubble sort for a fixed array of integers. (An array of integers can be defined as `int i[] = {1,2,3,4}`; the 2nd integer in an array can be printed using `printf("%d\n",i[1]);`.)
4. Modify your answer to (3) to sort characters into lexicographical order. (The 2nd character in a character array `i` can be printed using `printf("%c\n",i[1]);`.)