C and C++

1. Types — Variables — Expressions & Statements

Alastair R. Beresford
University of Cambridge
Lent Term 2007

Structure of this course

Programming in C:
- types, variables, expressions & statements
- functions, compilation, pre-processor
- pointers, structures
- extended examples, tick hints 'n' tips

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

Text books

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


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 *

* denotes CPL questions relevant to this course.
Where does the name ‘C’ come from?

- C was named as such since it was a successor to B
- B itself was a descendant of BCPL
- BCPL stood for Basic CPL
- CPL (Combined Programming Language) was a programming language developed between Cambridge & London
- Before London joined the project, ‘C’ stood for Cambridge

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

Classic first example

```c
#include <stdio.h>

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

Compile with:
```
$ cc example1.c
```

Execute program with:
```
$ ./a.out
Hello, world
```

```
Basic types

- C has a small and limited set of basic types:
  
<table>
<thead>
<tr>
<th>type</th>
<th>description (size)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>characters (≥ 8 bits)</td>
</tr>
<tr>
<td>int</td>
<td>integer values (≥ 16 bits, commonly one word)</td>
</tr>
<tr>
<td>float</td>
<td>single-precision floating point number</td>
</tr>
<tr>
<td>double</td>
<td>double-precision floating point number</td>
</tr>
</tbody>
</table>

- Precise size of types is architecture dependent
- Various type operators for altering type meaning, including: unsigned, long, short, const, static
- This means we can have types such as long int and unsigned char

Constants

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

<table>
<thead>
<tr>
<th>type</th>
<th>style</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>int</td>
<td>number, character or escape seq.</td>
<td>12 'A' \n' \007'</td>
</tr>
<tr>
<td>long int</td>
<td>number w/suffix 1 or L</td>
<td>1234L</td>
</tr>
<tr>
<td>float</td>
<td>number with '.', 'e' or 'E' and suffix f or F</td>
<td>1.234e3F or 1234.0f</td>
</tr>
<tr>
<td>double</td>
<td>number with '.', 'e' or 'E'</td>
<td>1.234e3 1234.0</td>
</tr>
<tr>
<td>long double</td>
<td>number 'l' or 'L'</td>
<td>1.234e31 or 1234.0L</td>
</tr>
</tbody>
</table>

- 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:
  ```c
  enum months {JAN=1,FEB,MAR}
  enum boolean {F,T,FALSE=0,TRUE, N=0,Y}
  ```
- Names 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 defined (i.e. storage set aside) exactly once
- 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 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:

<table>
<thead>
<tr>
<th>type</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic</td>
<td>+ - * / ++ -- %</td>
</tr>
<tr>
<td>logic</td>
<td>== != &gt; &gt;= &lt; &lt;=</td>
</tr>
<tr>
<td>bitwise</td>
<td>&amp;</td>
</tr>
<tr>
<td>assignment</td>
<td>+= -= *= /= %= &lt;&lt;= &gt;&gt;= &amp;=</td>
</tr>
</tbody>
</table>

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:
  int i = 1234;
  char c;
  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 automatic variables
  - 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)

Scope and type example

```c
#include <stdio.h>

int a; /*what value does a have? */
unsigned char b = 'A';
extern int alpha; /* safe to use this? */

int main(void) {
    extern unsigned char b; /* is this needed? */
    double a = 3.4;
    {
        extern a; /*why is this sloppy? */
        printf("%d %d\n",b,a+1); /*what will this print? */
    }
    return 0;
}
```

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 usually represented as an array of chars, terminated with a special character '\0'
- There is compiler support for string constants using the '"' character; for example:
  char str[]="two strs mer" "ged and terminated"
- String support is available in the string.h library
Control flow

- Control flow is similar to Java:
  - \( \text{exp} \ ? \ \text{exp} \ : \ \text{exp} \)
  - if (\( \text{exp} \)) \( \text{stmt}_1 \) else \( \text{stmt}_2 \)
  - switch(\( \text{exp} \)) {
    case \( \text{exp}_1 \):
      stmt\(_1\)
      ...
    default:
      stmt\(_{n+1}\)
  }
  - while (\( \text{exp} \)) \( \text{stmt} \)
  - for (\( \text{exp}_1; \ \text{exp}_2; \ \text{exp}_3 \)) \( \text{stmt} \)
  - do \( \text{stmt} \) while (\( \text{exp} \));
- The jump statements break and continue also exist

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:
  ```c
  for (...) {
    for (...) {
      ...
      if (critical_problem)
        goto error;
    }
  }
  ...
  
  error:
  fix problem, or abort
  ```

Control flow and string example

```c
#include <stdio.h>
#include <string.h>

char s[]="University of Cambridge Computer Laboratory";

int main(void) {

  char c;
  int i, j;

  for (i=0, j=strlen(s)-1; i<j; i++, j--) /* strlen(s)-1 ? */
    c=s[i], s[i]=s[j], s[j]=c;

  printf("%s\n",s);
  return 0;
}
```

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]);)