Programming in C and C++

Types, Variables, Expressions and Statements

Neel Krishnaswami and Alan Mycroft

Course Structure

Basics of C:

- Types, variables, expressions and statements
- Functions, compilation and the pre-processor
- Pointers and structures
- C programming tick hints and tips

C Programming Techniques:

- Pointer manipulation: linked lists, trees, and graph algorithms
- Memory management strategies: ownership and lifetimes, reference counting, tracing, and arenas
- Cache-aware programming: array-of-struct to struct-of-array transformations, blocking loops, intrusive data structures
- Unsafe behaviour and mitigations: eg, valgrind, asan, ubsan

Course Structure, continued

Course organization:

- Remaining lectures will be recorded and posted online
- Course hours will be in a lab format
- We will meet in the Intel lab during lecture hours for a programming exercise (unmarked, but will be used for supervisions)
- Virtual machine image with Linux and gcc installed available from course website

Introduction to C++:

- Final 2 C++ lectures will be traditional lecture format
- Similarities and differences from C
- Extensions in C++: templates, classes, memory allocation

Textbooks

Recommendations for C:

- *The C Programming Language*. Brian W. Kernighan and Dennis M. Ritchie.
- *C: A Reference Manual*. Samuel P. Harbison and Guy L. Steele.

The majority of the class will be on C, but here are two recommendations for C++ as well:

- *The C++ Programming Language*. Bjarne Stroustrup.
- *Thinking in C++: : Introduction to Standard C++*. Bruce Eckel.

The History of C++

- 1966: Martin Richards develops BCPL
- 1969: Ken Thompson designs B
- 1972: Dennis Ricthie designs C
- 1979: Bjarne Stroustrup designs C with Classes
- 1983: C with Classes becomes C++
- 1989: Original C90 ANSI C standard (ISO 1990)
- 1998: ISO C++ standard
- 1999: C99 standard (ISO 1999, ANSI 2000)
- 2011: C++11 ISO standard, C11 ISO standard
- 2014, 2017: C++ standard updates
- 2020: C++20 standard expected

C is a low-level, unsafe language

- C's primitive types are characters, numbers and addresses
- Operators work on these types
- No primitives on composite types (eg, strings, arrays, sets)
- Only static definition and stack-based locals built in (the heap is implemented as a library)
- I/O and threading are also implemented as libraries (using OS primitives)
- The language is *unsafe*: many erroneous uses of C features are not checked (either statically or at runtime), so errors can silently cause memory corruption and arbitrary code execution

The Classic First Program

```c
#include <stdio.h>

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

Compile with

```
$ cc example1.c
```

Execute with:

```
$ ./a.out
Hello, world!
$
```

Generate assembly with

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

Basic Types

- C has a small set of basic types

type	description
`char`	characters ($\geq$ 8 bits)
`int`	integers ($\geq$ 16 bits, usually 1 word)
`float`	single-precision floating point number
`double`	double-precision floating point number

- Precise size of types is architecture-dependent
- Various *type operators* alter meaning, including: `unsigned`, `short`, `long`, `const`, `volatile`
- This lets us make types like `long int` and `unsigned char`
- C99 added fixed-size types `int16_t`, `unit64_t` etc.

Constants

- Numeric literals can be written in many ways:

type	style	example
`char`	*none*	*none*
`int`	number, character or escape code	`12 'a' '\n'`
`long int`	num w/ suffix `l` or `L`	`1234L`
`float`	num with `'.'`, `'e'`, or `'E'` and suffix `'f'` or `'F'`	`1.234e3F 1234.0f`
`double`	num with `'.'`, `'e'`, or `'E'`	`1.234e3 1234.0`
`long double`	num with `'.'`, `'e'`, or `'E'` and suffix `'l'` or `'L'`	`1.23E3l 123.0L`

- Numbers can be expressed in octacl with '0' prefix and hexadecimal with '0x' prefix: `52 = 064 = 0x34`

Defining Constant Values

- An *enumeration* can specify a set of constants:
  `enum boolean {TRUE, FALSE}`
- Enumeration default to allocating successive integers from 0
- It is possible to assign values to constants

  ```
  enum months {JAN=1, FEB, MAR};
  enum boolean {F,T,FALSE=0,TRUE, N=0, Y};
  ```
- *Names* in an enumeration must be distinct, but values need not be.

Variables

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

Operators

- All operators (including assignment) return a result
- 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 different type
- Generally, conversion “widens” a value (e.g., `short` → `int`)
- However, “narrowing” is possible and may not generate a warning:

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

- Type conversion can be forced via 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: e.g. `x *= y - z`
- Every expression (even assignment) has a type and result
- Operator precedence gives an unambiguous parse for every expresion
- An expression (e.g., `x = 0`) becomes a *statement* when followed by a semicolon (i.e., `x = 0;`)
- Several expression can be separated using a comma ',' and expressions are then evaluated left-to-right: e.g., `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 and Compound Statements

- A *block* or *compound statement* is formed when multiple statementsare surrounded with braces (e.g. `{s1; s2; s3;}`)
- A block of statements is then equivalent to a single statement
- In C90, variables can only be declared or defined at the start of a block, but this restriction was lifted in C99
- Blocks are usually used in function definitions or control flow statements, but can appear anywhere a statement can

Variable Definition vs Declaration

- A variable can be *declared* without defining it 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

```
#include <stdio.h>

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

int main(void) {
  extern unsigned char b;       /* is this needed? */
  double a = 3.4;
  {
    extern a;                   /* 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 represented as an array of `char` terminated with the special character `'\0'`
- There is language support for this string representation in string contstants with double-quotes; for example `char s[]="two strings mer" "ged and terminated"` (note the implicit concatenation of string literals)
- String functions are 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
      ...
      case expn : stmtn
      default   : default_stmt
    }
    ```
  - `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

```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;
}
```

Goto (often considered harmful)

- The `goto` statement is never *required*
- It often results in difficult-to-understand code
- Exception handling (where you wish to exit from two or more loops) is one case where `goto` may be justified:

```
for (...) {
  for (...) {
    ...
    if (big_error) goto error;
  }
}
...
error: // handle error here
```