C and C++

1. Types — Variables — Expressions & Statements

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford)

Lent Term 2009
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
- 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 *
- 2008 Paper 3 Question 3 2008 Paper 10 Question 4

* 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
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

```
#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 ($\geq 8$ bits)</td>
</tr>
<tr>
<td>int</td>
<td>integer values ($\geq 16$ 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>
</tbody>
</table>
| int            | number, character or escape seq.           | 12 'A' '
' '\007' |
| long int       | number w/suffix l or L                     | 1234L            |
| float          | number with ‘.’, ‘e’ or ‘E’ and suffix f or F | 1.234e3F or 1234.0f |
| double         | number with ‘.’, ‘e’ or ‘E’                | 1.234e3 1234.0   |
| long double    | number ‘.’, ‘e’ or ‘E’ and suffix l or L   | 1.234E31 or 1234.0L |

- **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 *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></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:

```java
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 *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:
  ```c
  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:
  - `exp ? exp : exp`
  - `if (exp) stmt_1 else stmt_2`
  - `switch(exp) {
      case exp_1:
        stmt_1
      ...
      default:
        stmt_{n+1}
    }
  - `while (exp) stmt`
  - `for (exp_1; exp_2; exp_3) 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 (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
```
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]);.)
C and C++

2. Functions — Preprocessor

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford)

Lent Term 2009
Functions

- C does not have objects, but does have function support
- A function definition has a return type, parameter specification, and a body or statement; for example:
  ```c
  int power(int base, int n) { stmt }
  ```
- A function declaration has a return type and parameter specification followed by a semicolon; for example:
  ```c
  int power(int base, int n);
  ```
  - The use of the `extern` keyword for function declarations is optional
- All arguments to a function are copied, i.e. passed-by-value; modification of the local value does not affect the original
- Just as for variables, a function must have exactly one definition and can have multiple declarations
- A function which is used but only has a declaration, and no definition, results in a link error (more on this later)
- Functions cannot be nested
Function type-system nasties

- A function definition with no values (e.g. `power()`) is not an empty parameter specification, rather it means that its arguments should not be type-checked! (this is not the case in C++)
- Instead, a function with no arguments is declared using `void`
- An ellipsis (…) can be used for partial parameter specification, for example:
  ```c
  int printf(char* fmt,...) { stmt }
  ```
- The ellipsis is useful for defining functions with variable length arguments, but leaves a hole in the type system (`stdarg.h`)
- In comparison, C++ uses operator overloading to provide better I/O type safety (more on this later)
Recursion

- Functions can call themselves recursively
- On each call, a new set of local variables are created
- Therefore, a function recursion of depth $n$ has $n$ sets of variables
- Recursion can be useful when dealing with recursively defined data structures, like trees (more on such data structures later)
- Recursion can also be used as you would in ML:

```c
1 unsigned int fact(unsigned int n) {
2    return n ? n*fact(n-1) : 1;
3 }
```
Compilation

- A compiler transforms a C source file or execution unit into an object file.
- An object file consists of machine code, and a list of:
  - defined or exported symbols representing defined function names and global variables
  - undefined or imported symbols for functions and global variables which are declared but not defined
- A linker combines several object files into an executable by:
  - combining all object code into a single file
  - adjusting the absolute addresses from each object file
  - resolving all undefined symbols

The Part 1B Compiler Course describes how to build a compiler and linker in more detail.
Handling code in multiple files in C

- C separates declaration from definition for both variables and functions
- This allows portions of code to be split across multiple files
- Code in different files can then be compiled at different times
  - This allows libraries to be compiled once, but used many times
  - It also allows companies to sell binary-only libraries
- In order to use code written in another file we still need a declaration
- A header file can be used to:
  - supply the declarations of function and variable definitions in another file
  - provide preprocessor macros (more on this later)
  - avoid duplication (and :: errors) that would otherwise occur
- You might find the Unix tool nm useful for inspecting symbol tables
Multi-source file example

Header File — example4.h
1 /reverse a string in place */
2 void reverse(char str[]);
Variable and function scope with static

- The **static** keyword limits the scope of a variable or function
- In the global scope, **static** does not export the function or variable symbol
  - This prevents the variable or function from being called externally
- In the local scope, a **static** variable retains its value between function calls
  - A single static variable exists even if a function call is recursive
C Preprocessor

- The preprocessor is executed before any compilation takes place
- It manipulates the textual content of the source file in a single pass
- Amongst other things, the preprocessor:
  - deletes each occurrence of a backslash followed by a newline;
  - replaces comments by a single space;
  - replaces definitions, obeys conditional preprocessing directives and expands macros; and
  - it replaces escaped sequences in character constants and string literals and concatenates adjacent string literals
Controlling the preprocessor programmatically

- The preprocessor can be used by the programmer to rewrite source code
- This is a powerful (and, at times, useful) feature, but can be hard to debug (more on this later)
- The preprocessor interprets lines starting with `#` with a special meaning
- Two text substitution directives: `#include` and `#define`
- Conditional directives: `#if`, `#elif`, `#else` and `#endif`
The `#include` directive

- The `#include` directive performs text substitution.
- It is written in one of two forms:

  ```
  #include "filename"
  #include <filename>
  ```

- Both forms replace the `#include ...` line in the source file with the contents of `filename`.
- The quote (""") form searches for the file in the same location as the source file, then searches a predefined set of directories.
- The angle (<>) form searches a predefined set of directories.
- When a `#included` file is changed, all source files which depend on it should be recompiled.
The `#define` directive

- The `#define` directive has the form:
  
  ```
  #define name replacement text
  ```

- The directive performs a direct text substitution of all future examples of `name` with the `replacement text` for the remainder of the source file.

- The `name` has the same constraints as a standard C variable name.

- Replacement does not take place if `name` is found inside a quoted string.

- By convention, `name` tends to be written in upper case to distinguish it from a normal variable name.
Defining macros

► The `#define` directive can be used to define *macros* as well; for example: `#define MAX(A,B) ((A)>(B)?(A):(B))`

► In the body of the macro:
  ▶ prefixing a parameter in the replacement text with ‘#’ places the parameter value inside string quotes ("")
  ▶ placing ‘##’ between two parameters in the replacement text removes any whitespace between the variables in generated output

► Remember: the preprocessor only performs text substitution
  ▶ This means that syntax analysis and type checking doesn’t occur until the compilation stage
  ▶ This can, initially at least, generate some confusing compiler warnings on line numbers where the macro is used, rather than when it is defined; for example:
    `#define JOIN(A,B) (A ## B)"
Example

```c
#include <stdio.h>

#define PI 3.141592654
#define MAX(A,B) ((A)>(B)?(A):(B))
#define PERCENT(D) (100*D)        /* Wrong? */
#define DPRINT(D) printf(#D " = %.0g\n",D)
#define JOIN(A,B) (A ## B)

int main(void) {
    const unsigned int a1=3;
    const unsigned int i = JOIN(a,1);
    printf("%u %.0g\n",i, MAX(PI,3.14));
    DPRINT(MAX(PERCENT(0.32+0.16),PERCENT(0.15+0.48)));

    return 0;
}
```
Conditional preprocessor directives

Conditional directives: `#if`, `#ifdef`, `#ifndef`, `#elif` and `#endif`

- The preprocessor can use conditional statements to include or exclude code in later phases of compilation.

- `#if` accepts a (somewhat limited) integer expression as an argument and only retains the code between `#if` and `#endif` (or `#elif`) if the expression evaluates to a non-zero value; for example:

  ```
  #if SOME_DEF > 8 && OTHER_DEF != THIRD_DEF
  ```

- The built-in preprocessor function `defined` accepts a name as it’s sole argument and returns 1L if the name has been `#defined`; 0L otherwise.

- `#ifdef N` and `#ifndef N` are equivalent to `#if defined(N)` and `#if !defined(N)` respectively.

- `#undef` can be used to remove a `#defined` name from the preprocessor macro and variable namespace.
Example

Conditional directives have several uses, including preventing double definitions in header files and enabling code to function on several different architectures; for example:

```c
1 #if SYSTEM_SYSV
2 #define HDR "sysv.h"
3 #elif SYSTEM_BSD
4 #define HDR "bsd.h"
5 #else
6 #define HDR "default.h"
7 #endif
8 #include HDR
```

```c
1 ifndef MYHEADER_H
2 #define MYHEADER_H 1
3 ...
4 /* declarations & defns */
5 ...
6 #endif /* !MYHEADER_H */
```
Error control

- To help other compilers which generate C code (rather than machine code) as output, compiler line and filename warnings can be overridden with:
  
  ```
  #line constant "filename"
  ```

- The compiler then adjusts its internal value for the next line in the source file as `constant` and the current name of the file being processed as `filename` ("filename" may be omitted)

- The statement "`#error some text`" causes the preprocessor to write a diagnostic message containing `some text`

- There are several predefined identifiers that produce special information: `__LINE__`, `__FILE__`, `__DATE__`, and `__TIME__`.
Exercises

1. Write a function definition which matches the declaration
   \[\text{int cntlower(char str[]);}\]. The implementation should return the
   number of lower-case letters in a string

2. Use function recursion to write an implementation of merge sort for a
   fixed array of integers; how much memory does your program use for
   a list of length \(n\)?

3. Define a macro \(\text{SWAP}(t,x,y)\) that exchanges two arguments of type \(t\)
   (K&R, Exercise 4-14)

4. Define a macro \(\text{SWAP}(x,y)\) that exchanges two arguments of the same
   type (e.g. \text{int} or \text{char}) \text{without using a temporary}
C and C++

3. Pointers — Structures

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford)

Lent Term 2009
Pointers

- Computer memory is often abstracted as a sequence of bytes, grouped into words
- Each byte has a unique address or index into this sequence
- The size of a word (and byte!) determines the size of addressable memory in the machine
- A *pointer* in C is a variable which contains the memory address of another variable (this can, itself, be a pointer)
- Pointers are declared or defined using an asterisk (*); for example:
  ```c
  char *pc; or int **ppi;
  ```
- The asterisk binds to the variable name, not the type definition; for example `char *pc, c;`
- A pointer does *not* necessarily take the same amount of storage space as the type it points to
Example
Manipulating pointers

- The value “pointed to” by a pointer can be “retrieved” or dereferenced by using the unary * operator; for example:
  
  ```c
  int *p = ...  
  int x = *p;
  ```

- The memory address of a variable is returned with the unary ampersand (&) operator; for example
  
  ```c
  int *p = &x;
  ```

- Dereferenced pointer values can be used in normal expressions; for example: *pi += 5; or (*pi)++
Example

```c
#include <stdio.h>

int main(void) {
    int x=1, y=2;
    int *pi;
    int **ppi;

    pi = &x; ppi = &pi;
    printf("%p, %p, %d=%d=%d\n", ppi, pi, x, *pi, **ppi);
    pi = &y;
    printf("%p, %p, %d=%d=%d\n", ppi, pi, y, *pi, **ppi);

    return 0;
}
```
Pointers and arrays

- A C array uses consecutive memory addresses without padding to store data.

- An array name (without an index) represents the memory address of the beginning of the array; for example:
  ```c
  char c[10];
  char *pc = c;
  ```

- Pointers can be used to "index" into any element of an array; for example:
  ```c
  int i[10];
  int *pi = &i[5];
  ```
Pointer arithmetic

- *Pointer arithmetic* can be used to adjust where a pointer points; for example, if `pc` points to the first element of an array, after executing `pc+=3`; then `pc` points to the fourth element.

- A pointer can even be dereferenced using array notation; for example `pc[2]` represents the value of the array element which is two elements beyond the array element currently pointed to by `pc`.

- In summary, for an array `c`, `*(c+i)=c[i]` and `c+i=&c[i]`.

- A pointer is a variable, but an array name is not; therefore `pc=c` and `pc++` are valid, but `c=pc` and `c++` are not.
Example

```c
#include <stdio.h>

int main(void) {
    char str[] = "A string."
    char *pc = str;

    printf("%c %c %c\n", str[0], *pc, pc[3]);
    pc += 2;
    printf("%c %c %c\n", *pc, pc[2], pc[5]);

    return 0;
}
```
Pointers as function arguments

- Recall that all arguments to a function are copied, i.e. *passed-by-value*; modification of the local value does not affect the original.
- In the second lecture we defined functions which took an array as an argument; for example `void reverse(char s[])`.
- Why, then, does `reverse` affect the values of the array after the function returns (i.e. the array values haven’t been copied)?
  - because `s` is a pointer to the start of the array.
- Pointers of any type can be passed as parameters and return types of functions.
- Pointers allow a function to alter parameters passed to it.
Example

- Compare `swp1(a,b)` with `swp2(&a,&b)`:

```c
1 void swp1(int x, int y) {
2     int temp = x;
3     x = y;
4     y = temp;
5 }
```

```c
1 void swp2(int *px, int *py) {
2     int temp = *px;
3     *px = *py;
4     *py = temp;
5 }
```
Arrays of pointers

- C allows the creation of arrays of pointers; for example:
  ```c
  int *a[5];
  ```
- Arrays of pointers are particularly useful with strings.
- An example is C support of command line arguments:
  ```c
  int main(int argc, char *argv[]) { ... }
  ```
- In this case `argv` is an array of character pointers, and `argc` tells the programmer the length of the array.
Example


argc: 3

argv[0] programe\0
argv[1] firstarg\0
argv[2] secondarg\0
argv[3] NULL
Multi-dimensional arrays

- Multi-dimensional arrays can be declared in C; for example:
  ```c
  int i[5][10];
  ```
- Values of the array can be accessed using square brackets; for example: `i[3][2]`
- When passing a two dimensional array to a function, the first dimension is not needed; for example, the following are equivalent:
  ```c
  void f(int i[5][10]) { ... }  
  void f(int i[][10]) { ... }  
  void f(int (*i)[10]) { ... }  
  ```
- In arrays with higher dimensionality, all but the first dimension must be specified
Pointers to functions

- C allows the programmer to use pointers to functions
- This allows functions to be passed as arguments to functions
- For example, we may wish to parameterise a sort algorithm on different comparison operators (e.g. lexicographically or numerically)
- If the sort routine accepts a pointer to a function, the sort routine can call this function when deciding how to order values
Example

```c
void sort(int a[], const int len,
          int (*compare)(int, int))
{
    int i, j, tmp;
    for (i = 0; i < len - 1; i++)
        for (j = 0; j < len - 1 - i; j++)
            if ((*compare)(a[j], a[j + 1]))
                tmp = a[j], a[j] = a[j + 1], a[j + 1] = tmp;
}

int inc(int a, int b) {
    return a > b ? 1 : 0;
}
```
Example

```c
#include <stdio.h>
#include "example8.h"

int main(void) {
    int a[] = {1, 4, 3, 2, 5};
    unsigned int len = 5;
    sort(a, len, inc); // or sort(a, len, &inc);

    int *pa = a; //C99
    printf("[");
    while (len--) {    printf("%d%s", *pa++, len? "":"" );
        printf("\n");
    return 0;
}
```
The void * pointer

- C has a “typeless” or “generic” pointer: void *p
- This can be a pointer to anything
- This can be useful when dealing with dynamic memory
- Enables “polymorphic” code; for example:

```c
1  sort(void *p, const unsigned int len,
2    int (*comp)(void *, void *));
```

- However this is also a big “hole” in the type system
- Therefore void * pointers should only be used where necessary
Structure declaration

- A structure is a collection of one or more variables
- It provides a simple method of abstraction and grouping
- A structure may itself contain structures
- A structure can be assigned to, as well as passed to, and returned from functions
- We declare a structure using the keyword `struct`
- For example, to declare a structure `circle` we write
  ```
  struct circle {int x; int y; unsigned int r;};
  ```
- Once declared, a structure creates a new type
Structure definition

- To define an instance of the structure `circle` we write
  ```
  struct circle c;
  ```

- A structure can also be initialised with values:
  ```
  struct circle c = {12, 23, 5};
  ```

- An automatic, or local, structure variable can be initialised by function call:
  ```
  struct circle c = circle_init();
  ```

- A structure can declared, and several instances defined in one go:
  ```
  struct circle {int x; int y; unsigned int r;} a, b;
  ```
Member access

- A structure member can be accessed using ‘.’ notation: `structname.member`; for example: `pt.x`
- Comparison (e.g. `pt1 > pt2`) is undefined
- Pointers to structures may be defined; for example:
  ```
  struct circle *pc
  ```
- When using a pointer to a struct, member access can be achieved with the ‘.’ operator, but can look clumsy; for example: `(*pc).x`
- Alternatively, the ‘->’ operator can be used; for example: `pc->x`
Self-referential structures

- A structure declaration can contain a member which is a pointer whose type is the structure declaration itself
- This means we can build recursive data structures; for example:

```c
struct tree {
    int val;
    struct tree *left;
    struct tree *right;
}
```

```c
struct link {
    int val;
    struct link *next;
}
```
Unions

- A union variable is a single variable which can hold one of a number of different types

- A union variable is declared using a notation similar to structures; for example: `union u { int i; float f; char c;};`

- The size of a union variable is the size of its largest member

- The type held can change during program execution

- The type retrieved must be the type most recently stored

- Member access to unions is the same as for structures (‘.’ and ‘->’)

- Unions can be nested inside structures, and vice versa
Bit fields

- Bit fields allow low-level access to individual bits of a word.
- Useful when memory is limited, or to interact with hardware.
- A bit field is specified inside a struct by appending a declaration with a colon (:) and number of bits; for example:
  ```c
  struct fields { int f1 : 2; int f2 : 3;};
  ```
- Members are accessed in the same way as for structs and unions.
- A bit field member does not have an address (no & operator).
- Lots of details about bit fields are implementation specific:
  - *word boundary overlap & alignment, assignment direction, etc.*
Example (adapted from K&R)

```c
1 struct { /* a compiler symbol table */
2     char *name;
3     struct {
4         unsigned int is_keyword : 1;
5         unsigned int isExtern : 1;
6         unsigned int isStatic : 1;
7         ...  
8     } flags;
9     int uType;
10    union {
11         int ival; /* accessed as symtab[i].u.ival */
12         float fVal;
13         char *sVal;
14     } u;
15  } symtab[NSYM];
```
Exercises

1. If p is a pointer, what does p[-2] mean? When is this legal?
2. Write a string search function with a declaration of
   char *strfind(const char *s, const char *f); which returns a
   pointer to first occurrence of s in f (and NULL otherwise)
3. If p is a pointer to a structure, write some C code which uses all the
   following code snippets: “++p->i”, “p++->i”, “*p->i”, “*p->i++”,
   “(*p->i)++” and “*p++->i”; describe the action of each code snippet
4. Write a program calc which evaluates a reverse Polish expression given on
   the command line; for example
   $ calc 2 3 4 + *
   should print 14 (K&R Exercise 5-10)
C and C++

4. Misc. — Library Features — Gotchas — Hints ’n’ Tips

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford)

Lent Term 2009
Example

int **ppi
int *pi
int i
char *pc
char c

0x2c  0x30  0x34  0x38
0x4c  0x50  0x60
Uses of const and volatile

- Any declaration can be prefixed with `const` or `volatile`
- A `const` variable can only be assigned a value when it is defined
- The `const` declaration can also be used for parameters in a function definition
- The `volatile` keyword can be used to state that a variable may be changed by hardware, the kernel, another thread etc.
  - For example, the `volatile` keyword may prevent unsafe compiler optimisations for memory-mapped input/output
- The use of pointers and the `const` keyword is quite subtle:
  - `const int *p` is a pointer to a `const int`
  - `int const *p` is also a pointer to a `const int`
  - `int *const p` is a `const` pointer to an `int`
  - `const int *const p` is a `const` pointer to a `const int`
Example

```c
int main(void) {
    int i = 42;
    int j = 28;

    const int *pc = &i;    // Also: "int const *pc"
    *pc = 41;
    // Wrong

    pc = &j;

    int *const cp = &i;
    *cp = 41;
    // Wrong

    cp = &j;

    const int *const cpc = &i;
    *cpc = 41;              // Wrong
    cpc = &j;               // Wrong
    return 0;
}
```
**Typedefs**

- The **typedef** operator, creates new data type names; for example, `typedef unsigned int Radius;`
- Once a new data type has been created, it can be used in place of the usual type name in declarations and casts; for example, `Radius r = 5; ...; r = (Radius) rshort;`
- A **typedef** declaration does *not* create a new type
  - It just creates a synonym for an existing type
- A **typedef** is particularly useful with structures and unions:

```c
1 typedef struct llist *llptr;
2 typedef struct llist {
3   int val;
4   llptr next;
5 } linklist;
```
In-line functions

- A function in C can be declared `inline`; for example:

  ```c
  1 inline fact(unsigned int n) {
  2     return n ? n*fact(n-1) : 1;
  3 }
  ```

- The compiler will then try to “in-line” the function
  - A clever compiler might generate 120 for `fact(5)`

- A compiler might not always be able to “in-line” a function

- An `inline` function must be `defined` in the same execution unit as it is used

- The `inline` operator does not change function semantics
  - the in-line function itself still has a unique address
  - static variables of an in-line function still have a unique address
That’s it!

- We have now explored most of the C language
- The language is quite subtle in places; in particular watch out for:
  - operator precedence
  - pointer assignment (particularly function pointers)
  - implicit casts between `int`s of different sizes and `char`
- There is also extensive standard library support, including:
  - shell and file I/O (`stdio.h`)
  - dynamic memory allocation (`stdlib.h`)
  - string manipulation (`string.h`)
  - character class tests (`ctype.h`)
  - ...
  - (Read, for example, K&R Appendix B for a quick introduction)
  - (Or type “`man function`” at a Unix shell for details)
Library support: I/O

I/O is not managed directly by the compiler; support in stdio.h:

- int printf(const char *format, ...);
- int sprintf(char *str, const char *format, ...);
- int scanf(const char *format, ...);

- FILE *fopen(const char *path, const char *mode);
- int fclose(FILE *fp);
- size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
- size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
- int fprintf(FILE *stream, const char *format, ...);
- int fscanf(FILE *stream, const char *format, ...);
```c
#include<stdio.h>
define BUFSIZE 1024

int main(void) {
    FILE *fp;
    char buffer[BUFSIZE];

    if ((fp=fopen("somefile.txt","rb")) == 0) {
        perror("fopen error:");
        return 1;
    }

    while(!feof(fp)) {
        int r = fread(buffer,sizeof(char),BUFSIZE,fp);
        fwrite(buffer,sizeof(char),r,stdout);
    }

    fclose(fp);
    return 0;
}
```
Library support: dynamic memory allocation

- Dynamic memory allocation is not managed directly by the C compiler
- Support is available in stdlib.h:
  - `void *malloc(size_t size)`
  - `void *calloc(size_t nobj, size_t size)`
  - `void *realloc(void *p, size_t size)`
  - `void free(void *p)`
- The C `sizeof` unary operator is handy when using `malloc`:
  - `p = (char *) malloc(sizeof(char)*1000)`
- Any successfully allocated memory must be deallocated manually
  - Note: `free()` needs the pointer to the allocated memory
- Failure to deallocate will result in a memory leak
# Gotchas: operator precedence

```c
#include<stdio.h>

struct test {int i;};
typedef struct test test_t;

int main(void) {

test_t a,b;
test_t *p[] = {&a,&b};
p[0]->i=0;
p[1]->i=0;
test_t *q = p[0];

printf("%d\n",++q->i); //What does this do?
return 0;
}
```
Gotchas: \texttt{i++}

```c
#include <stdio.h>

int main(void) {
    int i=2;
    int j=i++ + ++i;
    printf("%d %d\n",i,j); //What does this print?
    return 0;
}
```
#include <stdio.h>

char *unary(unsigned short s) {
    char local[s+1];
    int i;
    for (i=0;i<s;i++) local[i]=’1’;
    local[s]=’\0’;
    return local;
}

int main(void) {
    printf("%s\n",unary(6));  //What does this print?
    return 0;
}
# include <stdio.h>

char global[10];

char *unary(unsigned short s) {
    char local[s+1];
    char *p = s%2 ? global : local;
    int i;
    for (i=0;i<s;i++) p[i]=’1’;
    p[s]=’\0’;
    return p;
}

int main(void) {
    printf("%s\n", unary(6)); //What does this print?
    return 0;
}
Gotchas: careful with pointers

```c
#include <stdio.h>

struct values { int a; int b; };

int main(void) {
    struct values test2 = {2,3};
    struct values test1 = {0,1};

    int *pi = &(test1.a);
    pi += 1; //Is this sensible?
    printf("%d\n",*pi);
    pi += 2; //What could this point at?
    printf("%d\n",*pi);

    return 0;
}
```
Gotchas: XKCD pointers

Okay, human.

Huh?

Before you hit ‘compile’, listen up.

You know when you’re falling asleep, and you imagine yourself walking or something.

And suddenly you misstep, stumble, and jolt awake?

Yeah!

Well, that’s what a segfault feels like.

Double-check your damn pointers, okay?
Tricks: Duff’s device

```c
send(int *to, int *from, int count)
{
    int n=(count+7)/8;
    switch(count%8){
    case 0: do{  *to = *from++;  
               } while(--n>0);
    case 7:    *to = *from++;     
    case 6:    *to = *from++;     
    case 5:    *to = *from++;     
    case 4:    *to = *from++;     
    case 3:    *to = *from++;     
    case 2:    *to = *from++;     
    case 1:    *to = *from++;     
        }
}
```
Assessed exercise

- To be completed by midday on 24th April 2009
- Sign-up sheet removed midday on 24th April 2009
- Viva examinations 1300-1600 on 7th May 2009
- Viva examinations 1300-1600 on 8th May 2009
- Download the starter pack from:
  http://www.cl.cam.ac.uk/Teaching/current/CandC++/
- This should contain eight files:
  server.c  rfc0791.txt  message1  message3
  client.c  rfc0793.txt  message2  message4
Exercise aims

Demonstrate an ability to:

- Understand (simple) networking code
- Use control flow, functions, structures and pointers
- Use libraries, including reading and writing files
- Understand a specification
- Compile and test code

Task is split into three parts:

- Comprehension and debugging
- Preliminary analysis
- Completed code and testing
Exercise submission

- Assessment is in the form of a ‘tick’
- There will be a short viva; remember to sign up!
- Submission is via email to c-tick@cl.cam.ac.uk
- Your submission should include seven files, packed in to a ZIP file called crsid.zip and attached to your submission email:

  answers.txt  client1.c  summary.c  message1.txt  message2.jpg
  server1.c   extract.c
Hints: IP header

```
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---------------------------------------------+
|Version| IHL  |Type of Service|          Total Length          |   |
+---------------------------------------------+
|      |     |              | Identification | Flags| Fragment Offset |   |
+---------------------------------------------+
|      |     |              | Time to Live| Protocol| Header Checksum |   |
+---------------------------------------------+
|      |     |              |    |          | Source Address   |   |
+---------------------------------------------+
|      |     |              |    |          | Destination Address |   |
+---------------------------------------------+
|      |     |              |    |          | Options          | Padding |
+---------------------------------------------+
```
Hints: IP header (in C)

```c
#include <stdint.h>

struct ip {
    uint8_t hlenver;
    uint8_t tos;
    uint16_t len;
    uint16_t id;
    uint16_t off;
    uint8_t ttl;
    uint8_t p;
    uint16_t sum;
    uint32_t src;
    uint32_t dst;
};

#define IP_HLEN(lenver) (lenver & 0x0f)
#define IP_VER(lenver) (lenver >> 4)
```
Hints: network byte order

- The IP network is big-endian; x86 is little-endian
- Reading multi-byte values requires conversion
- The BSD API specifies:
  - `uint16_t ntohs(uint16_t netshort)`
  - `uint32_t ntohl(uint32_t netlong)`
  - `uint16_t htons(uint16_t hostshort)`
  - `uint32_t htonl(uint32_t hostlong)`
Exercises

1. What is the value of i after executing each of the following:
   1.1 i = sizeof(char);
   1.2 i = sizeof(int);
   1.3 int a; i = sizeof a;
   1.4 char b[5]; i = sizeof(b);
   1.5 char *c=b; i = sizeof(c);
   1.6 struct {int d;char e;} s; i = sizeof s;
   1.7 void f(int j[5]) { i = sizeof j;}
   1.8 void f(int j[][10]) { i = sizeof j;}

2. Use struct to define a data structure suitable for representing a binary tree of integers. Write a function heapify(), which takes a pointer to an integer array of values and a pointer to the head of an (empty) tree and builds a binary heap of the integer array values. (Hint: you’ll need to use malloc())

3. What other C data structure can be used to represent a heap? Would using this structure lead to a more efficient implementation of heapify()?
C and C++

5. Overloading — Namespaces — Classes

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford)

Lent Term 2009
C++

To quote Bjarne Stroustrup:

“C++ is a general-purpose programming language with a bias towards systems programming that:
  ▶ is a better C
  ▶ supports data abstraction
  ▶ supports object-oriented programming
  ▶ supports generic programming.”
C++ fundamental types

- C++ has all the fundamental types C has
  - character literals (e.g. 'a') are now of type `char`
- In addition, C++ defines a new fundamental type, `bool`
- A `bool` has two values: `true` and `false`
- When cast to an integer, `true`→`1` and `false`→`0`
- When casting from an integer, non-zero values become `true` and `false` otherwise
C++ enumeration

- Unlike C, C++ enumerations define a new type; for example:
  ```cpp
enum flag {is_keyword=1, is_static=2, isExtern=4, ... }
```

- When defining storage for an instance of an enumeration, you use its name; for example:
  ```cpp
  flag f = is_keyword
  ```

- Implicit type conversion is not allowed:
  ```cpp
  f = 5; //wrong    f = flag(5); //right
  ```

- The maximum valid value of an enumeration is the enumeration’s largest value rounded up to the nearest larger binary power minus one.

- The minimum valid value of an enumeration with no negative values is zero.

- The minimum valid value of an enumeration with negative values is the nearest least negative binary power.
References

- C++ supports references, which provide an alternative name for a variable
- Generally used for specifying parameters to functions and return values as well as overloaded operators (more later)
- A reference is declared with the & operator; for example:
  ```c++
  int i[] = {1,2}; int &refi = i[0];
  ```
- A reference must be initialised when it is defined
- A variable referred to by a reference cannot be changed after it is initialised; for example:
  ```c++
  refi++; //increments value referenced
  ```
References in function arguments

- When used as a function parameter, a referenced value is not copied; for example:
  
  ```c
  void inc(int& i) { i++;} //bad style?
  ```

- Declare a reference as `const` when no modification takes place

- It can be noticeably more efficient to pass a large struct by reference

- Implicit type conversion into a temporary takes place for a `const` reference but results in an error otherwise; for example:

  ```c
  1 float fun1(float&);
  2 float fun2(const float&);
  3 void test() {
  4    double v=3.141592654;
  5    fun1(v); //Wrong
  6    fun2(v);
  7 }
  ```
Overloaded functions

- Functions doing different things should have different names
- It is possible (and sometimes sensible!) to define two functions with the same name
- Functions sharing a name must differ in argument types
- Type conversion is used to find the “best” match
- A best match may not always be possible:

```c
1  void f(double);
2  void f(long);
3  void test() {
4       f(1L);  //f(long)
5       f(1.0); //f(double)
6       f(1);   //Wrong: f(long(1)) or f(double(1))?
```
Scoping and overloading

- Functions in different scopes are not overloaded; for example:

```c
void f(int);

void example() {
    void f(double);
    f(1); // calls f(double);
}
```
Default function arguments

- A function can have default arguments; for example:
  ```c
  double log(double v, double base=10.0);
  ```
- A non-default argument cannot come after a default; for example:
  ```c
  double log(double base=10.0, double v); // wrong
  ```
- A declaration does not need to name the variable; for example:
  ```c
  double log(double v, double=10.0);
  ```
- Be careful of the interaction between * and =; for example:
  ```c
  void f(char* = 0); // Wrong '*=0' is assignment
  ```
Namespaces

Related data can be grouped together in a namespace:

```cpp
namespace Stack { //header file
    void push(char);
    char pop();
}

namespace Stack { //implementation
    const int max_size = 100;
    char s[max_size];
    int top = 0;

    void push(char c) { ... }
    char pop() { ... }
}

void f() { //usage
    ...
    Stack::push('c');
    ...
}
```
Using namespaces

- A namespace is a *scope* and expresses *logical* program structure
- It provides a way of collecting together related pieces of code
- A namespace without a name limits the scope of variables, functions and classes within it to the local execution unit
- The same namespace can be declared in several source files
- The *global* function `main()` cannot be inside a namespace
- The use of a variable or function name from a different namespace must be qualified with the appropriate namespace(s)
  - The keyword `using` allows this qualification to be stated once, thereby shortening names
  - Can also be used to generate a hybrid namespace
  - `typedef` can be used: `typedef Some::Thing thing;`

- A namespace can be defined more than once
  - Allows, for example, internal and external library definitions
Example

```cpp
namespace Module1 { int x; }

namespace Module2 {
    inline int sqr(const int& i) {return i*i;}
    inline int halve(const int& i) {return i/2;}
}

using namespace Module1;  //"import" everything

int main() {
    using Module2::halve;  //"import" the halve function
    x = halve(x);
    sqr(x);    //Wrong
}
```
Linking C and C++ code

- The directive `extern "C"` specifies that the following declaration or definition should be linked as C, not C++ code:
  ```c
  extern "C" int f();
  ```
- Multiple declarations and definitions can be grouped in curly brackets:
  ```c
  extern "C" {
    int globalvar;  //definition
    int f();
    void g(int);
  }
  ```
Linking C and C++ code

- Care must be taken with pointers to functions and linkage:

```c
1 extern "C" void qsort(void* p, \
2     size_t nmemb, size_t size, \n3     int (*compar)(const void*, const void*));

4 int compare(const void*,const void*);

5 char s[] = "some chars";
6 qsort(s,9,1,compare); //Wrong
```
User-defined types

- C++ provides a means of defining classes and instantiating objects
- Classes contain both data storage and functions which operate on storage
- Classes have access control:
  - private, protected and public
- Classes are created with `class` or `struct` keywords
  - `struct` members default to `public` access; `class` to `private`
- A member function with the same name as a `class` is called a `constructor`
- A member function with the same name as the class, prefixed with a tilde (~), is called a `destructor`
- A constructor can be overloaded to provide multiple instantiation methods
- Can create `static` (i.e. per `class`) member variables
Example

```cpp
class Complex {
    double re, im;
    public:
        Complex(double r = 0.0L, double i = 0.0L);
};

Complex::Complex(double r, double i) {
    re = r, im = i;
}

int main() {
    Complex c(2.0), d(), e(1, 5.0L);
    return 0;
}
```
Constructors and destructors

- A default constructor is a function with no arguments (or only default arguments)
- If no constructor is specified, the compiler will generate one
- The programmer can specify one or more constructors
- Only one constructor is called when an object is created
- There can only be one destructor
  - This is called when a stack allocated object goes out of scope or when a heap allocated object is deallocated with delete; this also occurs for stack allocated objects during exception handling (more later)
Copy constructor

- A new class instance can defined by assignment; for example:
  
  ```
  Complex c(1,2);
  Complex d = c;
  ```

- In this case, the new class is initialised with copies of all the existing class’ non-static member variables; no constructor is called

- This behaviour may not always be desirable (e.g. consider a class with a pointer as a member variable)
  
  - In which case, define an alternative copy constructor:
    ```
    Complex::Complex(const Complex&) { ... }
    ```

- If a copy constructor is not appropriate, make the copy constructor a private member function
Assignment operator

- By default a class is copied on assignment by over-writing all non-static member variables; for example:

```cpp
1 Complex c(), d(1.0,2.3);
2 c = d; //assignment
```

- This behaviour may also not be desirable

- The assignment operator (operator=) can be defined explicitly:

```cpp
1 Complex& Complex::operator=(const Complex& c) {
2   ...
3 }
```
Constant member functions

- Member functions can be declared `const`
- Prevents object members being modified by the function:

```cpp
1 double Complex::real() const {
2     return re;
3 }
```
Arrays and the free store

- An array of class objects can be defined if a class has a default constructor
- C++ has a `new` operator to place items on the heap:
  ```
  Complex* c = new Complex(3.4);
  ```
- Items on the heap exist until they are explicitly deleted:
  ```
  delete c;
  ```
- Since C++ (like C) doesn’t distinguish between a pointer to an object and a pointer to an array of objects, array deletion is different:
  ```
  1 Complex* c = new Complex[5];
  2   ...
  3 delete[] c; //Cannot use "delete" here
  ```
- When an object is deleted, the object destructor is invoked
Exercises

1. Write an implementation of a class `LinkList` which stores zero or more positive integers internally as a linked list on the heap. The class should provide appropriate constructors and destructors and a method `pop()` to remove items from the head of the list. The method `pop()` should return -1 if there are no remaining items. Your implementation should override the copy constructor and assignment operator to copy the linked-list structure between class instances. You might like to test your implementation with the following:

```c
1 int main() {
2   int test[] = {1,2,3,4,5};
3   LinkList l1(test+1,4), l2(test,5);
4   LinkList l3=l1, l4;
5   l4=l1;
6   printf("%d %d %d\n",l1.pop(),l3.pop(),l4.pop());
7   return 0;
8 }
```

*Hint: heap allocation & deallocation should occur exactly once!*
C and C++

6. Operators — Inheritance — Virtual

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford)

Lent Term 2009
From last lecture ...

```cpp
class Complex {
    double re, im;

public:
    Complex(double r=0.0L, double i=0.0L);
};

Complex::Complex(double r, double i) {
    re = r, im = i;
}

int main() {
    Complex c(2.0), d(), e(1, 5.0L);
    return 0;
}
```
Operators

- C++ allows the programmer to overload the built-in operators

- For example, a new test for equality:

```cpp
1 bool operator==(Complex a, Complex b) {
2     return a.real() == b.real()
3         && a.imag() == b.imag();
4 }
```

- An operator can be defined or declared within the body of a class, and in this case one fewer argument is required; for example:

```cpp
1 bool Complex::operator==(Complex b) {
2     return re == b.real() && im == b.imag();
3 }
```

- Almost all operators can be overloaded
Streams

- Overloaded operators also work with built-in types
- Overloading is used to define a C++ "printf"; for example:

```cpp
#include <iostream>

int main() {
    const char* s = "char array";

    std::cout << s << std::endl;

    // Unexpected output; prints &s[0]
    std::cout.operator<<(s).operator<<(std::endl);

    // Expected output; prints s
    std::operator<<(std::cout, s);
    std::cout.operator<<(std::endl);
    return 0;
}
```
The ‘this’ pointer

- If an operator is defined in the body of a class, it may need to return a reference to the current object
  - The keyword \texttt{this} can be used
- For example:

```cpp
Complex& Complex::operator+=(Complex b) {
    re += b.real();
    this->im += b.imag();
    return *this;
}
```
Class instances as member variables

- A class can have an instance of another class as a member variable
- How can we pass arguments to the class constructor?
- New notation for a constructor:

```cpp
1 class X {
2    Complex c;
3    Complex d;
4    X(double a, double b): c(a,b), d(b) {
5        ...
6    }
7};
```

- This notation must be used to initialise const and reference members
- It can also be more efficient
Temporary objects

- Temporary objects are often created during execution
- A temporary which is not bound to a reference or named object exists only during evaluation of a *full expression*
- Example: the `string` class has a function `c_str()` which returns a pointer to a C representation of a string:

```cpp
1  string a("A "), b("string");
2  const char *s1 = a.c_str();    //Okay
3  const char *s2 = (a+b).c_str(); //Wrong
4    ...
5  //s still in scope here, but the temporary holding
6  //"a+b" has been deallocated
7    ...
8  string tmp = a+b;
9  const char *s3 = tmp.c_str();  //Okay
```
A (non-member) **friend** function can access the private members of a class instance it befriends

This can be done by placing the function declaration inside the class definition and prefixing it with the keyword **friend**; for example:

```cpp
class Matrix {
...
friend Vector operator*(const Matrix&, 
    const Vector&);
...
};
```
Inheritance

- C++ allows a class to inherit features of another:

```cpp
class vehicle {
    int wheels;
public:
    vehicle(int w=4):wheels(w) {} 
};

class bicycle : public vehicle {
    bool panniers;
public:
    bicycle(bool p):vehicle(2), panniers(p) {} 
};

int main() {
    bicycle(false);
}
```
Derived member function call

- **Default** derived member function call semantics differ from Java:

```cpp
1 class vehicle {
2   int wheels;
3 public:
4   vehicle(int w=4):wheels(w) {}
5   int maxSpeed() {return 60;}
6 };

7
class bicycle : public vehicle {
9   int panniers;
10 public:
11   bicycle(bool p=true):vehicle(2),panniers(p) {}
12   int maxSpeed() {return panniers ? 12 : 15;}
13 };
```
Example

```cpp
#include <iostream>
#include "example13.hh"

void print_speed(vehicle &v, bicycle &b) {
    std::cout << v.maxSpeed() << " ";
    std::cout << b.maxSpeed() << std::endl;
}

int main() {
    bicycle b = bicycle(true);
    print_speed(b,b); //prints "60 12"
}
```
Virtual functions

- Non-virtual member functions are called depending on the *static type* of the variable, pointer or reference
- Since a derived class can be cast to a base class, this prevents a derived class from overloading a function
- To get polymorphic behaviour, declare the function *virtual* in the superclass:

```cpp
class vehicle {
    int wheels;
    public:
    vehicle(int w=4):wheels(w) {} 
    virtual int maxSpeed() {return 60;} 
};
```
Virtual functions

- In general, for a virtual function, selecting the right function has to be run-time decision; for example:

```cpp
1  bicycle b(true);
2  vehicle v;
3  vehicle* pv;
4
5  user_input() ? pv = &b : pv = &v;
6
7  std::cout << pv->maxSpeed() << std::endl;
8 }
```
Enabling virtual functions

- To enable virtual functions, the compiler generates a virtual function table or vtable
- A vtable contains a pointer to the correct function for each object instance
- The vtable is an example of indirection
- The vtable introduces run-time overhead
Abstract classes

- Sometimes a base class is an un-implementable concept
- In this case we can create an abstract class:

```cpp
class shape {
  public:
  virtual void draw() = 0;
}
```

- It is not possible to instantiate an abstract class:
  ```cpp
  shape s; //Wrong
  ```
- A derived class can provide an implementation for some (or all) the abstract functions
- A derived class with no abstract functions can be instantiated
Example

```cpp
1 class shape {
2 public:
3     virtual void draw() = 0;
4 }

6 class circle : public shape {
7 public:
8     //...
9     void draw() { /* impl */ }
10 }
```
Multiple inheritance

- It is possible to inherit from multiple base classes; for example:
  
  ```cpp
  class ShapelyVehicle: public vehicle, public shape {
      ...
  }
  ```

- Members from both base classes exist in the derived class
- If there is a name clash, explicit naming is required
- This is done by specifying the class name; for example:
  ```cpp
  ShapelyVehicle sv;
  sv.vehicle::maxSpeed();
  ```
Multiple instances of a base class

- With multiple inheritance, we can build:

```cpp
1 class A {};
2 class B : public A {};
3 class C : public A {};
4 class D : public B, public C {};
```

- This means we have two instances of `A` even though we only have a single instance of `D`.

- This is legal C++, but means all references to `A` must be stated explicitly:

```cpp
1 D d;
2 d.B::var=3;
3 d.C::var=4;
```
Virtual base classes

- Alternatively, we can have a *single* instance of the base class
- Such a “virtual” base class is shared amongst all those deriving from it

```cpp
1  class Vehicle { int VIN; };  
2  class Boat : public virtual Vehicle { ... };  
3  class Car : public virtual Vehicle { ... };  
4  class JamesBondCar : public Boat, public Car { ... }; 
```
Exercises

1. If a function $f$ has a static instance of a class as a local variable, when might the class constructor be called?

2. Write a class `Matrix` which allows a programmer to define two dimensional matrices. Overload the common operators (e.g. `+`, `-`, `*`, and `/`)

3. Write a class `Vector` which allows a programmer to define a vector of length two. Modify your `Matrix` and `Vector` classes so that they interoperate correctly (e.g. $v2 = m*v1$ should work as expected)

4. Why should destructors in an abstract class almost always be declared `virtual`?
C and C++

7. Exceptions — Templates

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford)

Lent Term 2009
Exceptions

- Some code (e.g. a library module) may detect an error but not know what to do about it; other code (e.g. a user module) may know how to handle it.
- C++ provides exceptions to allow an error to be communicated.
- In C++ terminology, one portion of code *throws* an exception; another portion *catches* it.
- If an exception is thrown, the call stack is unwound until a function is found which catches the exception.
- If an exception is not caught, the program terminates.
Throwing exceptions

- Exceptions in C++ are just normal values, matched by type
- A class is often used to define a particular error type:
  ```
  class MyError {}; 
  ```
- An instance of this can then be thrown, caught and possibly re-thrown:
  ```
  1 void f() { ... throw MyError();; ... }
  2 ...
  3 try {
  4    f();
  5  }
  6 catch (MyError) {
  7    //handle error
  8    throw; //re-throw error
  9  }
  ```
Conveying information

The “thrown” type can carry information:

```c
struct MyError {
    int errorcode;
    MyError(i):errorcode(i) {}
};

void f() { ... throw MyError(5); ... }

try {
    f();
} catch (MyError x) {
    //handle error (x.errorcode has the value 5)
    ... 
}
```
Handling multiple errors

- Multiple catch blocks can be used to catch different errors:

```java
try {
    ...
}
catch (MyError x) {
    //handle MyError
}
catch (YourError x) {
    //handle YourError
}
```

- Every exception will be caught with `catch(...)`
- Class hierarchies can be used to express exceptions:
```cpp
#include <iostream>

struct SomeError {virtual void print() = 0;};
struct ThisError : public SomeError {
    virtual void print() {
        std::cout << "This Error" << std::endl;
    }
};
struct ThatError : public SomeError {
    virtual void print() {
        std::cout << "That Error" << std::endl;
    }
};

int main() {
    try { throw ThisError(); } 
    catch (SomeError& e) { //reference, not value
        e.print();
    }
    return 0;
}
```
Exceptions and local variables

- When an exception is thrown, the stack is unwound
- The destructors of any local variables are called as this process continues
- Therefore it is good C++ design practise to wrap any locks, open file handles, heap memory etc., inside a stack-allocated class to ensure that the resources are released correctly
Templates

- Templates support *meta-programming*, where code can be evaluated at compile-time rather than run-time
- Templates support *generic programming* by allowing types to be parameters in a program
- Generic programming means we can write one set of algorithms and one set of data structures to work with objects of *any* type
- We can achieve some of this flexibility in C, by casting everything to `void *` (e.g. `sort` routine presented earlier)
- The C++ Standard Template Library (STL) makes extensive use of templates
An example: a stack

- The stack data structure is a useful data abstraction concept for objects of many different types
- In one program, we might like to store a stack of `ints`
- In another, a stack of `NetworkHeader` objects
- Templates allow us to write a single `generic` stack implementation for an unspecified type `T`
- What functionality would we like a stack to have?
  - `bool isEmpty();`
  - `void push(T item);`
  - `T pop();`
  - `...`
- Many of these operations depend on the type `T`
Creating a stack template

- A class template is defined as:

```cpp
1 template<class T> class Stack {
2     ...
3 }
```

- Where `class T` can be any C++ type (e.g. `int`)
- When we wish to create an instance of a `Stack` (say to store `ints`) then we must specify the type of `T` in the declaration and definition of the object: `Stack<int> intstack;`
- We can then use the object as normal: `intstack.push(3);`
- So, how do we implement `Stack`?
  - Write `T` whenever you would normally use a concrete type
template<class T> class Stack {

    struct Item { //class with all public members
        T val;
        Item* next;
        Item(T v) : val(v), next(0) {}
    };

    Item* head;

    Stack(const Stack& s) {} //private
    Stack& operator=(const Stack& s) {} //

public:
    Stack() : head(0) {}
    ~Stack();
    T pop();
    void push(T val);
    void append(T val);
};
#include "example16.hh"

template<class T> void Stack<T>::append(T val) {
    Item **pp = &head;
    while(*pp) {pp = &((*pp)->next);}    
    *pp = new Item(val);
}

//Complete these as an exercise
template<class T> void Stack<T>::push(T) {  /* ... */}
template<class T> T Stack<T>::pop() { /* ... */}
template<class T> Stack<T>::~Stack() { /* ... */}

int main() {
    Stack<char> s;
    s.push('a'), s.append('b'), s.pop();
}
Template details

- A template parameter can take an integer value instead of a type:
  ```cpp
template<int i> class Buf { int b[i]; ... };
```
- A template can take several parameters:
  ```cpp
template<class T, int i> class Buf { T b[i]; ... };
```
- A template can even use one template parameter in the definition of a subsequent parameter:
  ```cpp
template<class T, T val> class A { ... };
```
- A templated class is not type checked until the template is instantiated:
  ```cpp
template<class T> class B {const static T a=3;};
  
  B<int> b; is fine, but what about B<B<int> > bi;?
```
- Template definitions often need to go in a header file, since the compiler needs the source to instantiate an object
Default parameters

- Template parameters may be given default values

```cpp
1   template <class T, int i=128> struct Buffer{
2       T buf[i];
3   };

4
5   int main() {
6       Buffer<int> B; //i=128
7       Buffer<int,256> C;
8   }
```
Specialization

- The class `T` template parameter will accept any type `T`
- We can define a *specialization* for a particular type as well:

```cpp
#include <iostream>
class A {}

template<class T> struct B {
  void print() { std::cout << "General" << std::endl; }
};
template<> struct B<A> {
  void print() { std::cout << "Special" << std::endl; }
};

int main() {
  B<A> b1;
  B<int> b2;
  b1.print(); //Special
  b2.print(); //General
}
```
Templated functions

- A function definition can also be specified as a template; for example:

```
1 template<class T> void sort(T a[],
2     const unsigned int& len);
```

- The type of the template is inferred from the argument types:
```
int a[] = {2,1,3}; sort(a,3); \implies T is an int
```
- The type can also be expressed explicitly:
```
sort<int>(a)
```
- There is no such type inference for templated classes
- Using templates in this way enables:
  - better type checking than using `void *`
  - potentially faster code (no function pointers)
  - larger binaries if `sort()` is used with data of many different types
#include <iostream>

template<class T> void sort(T a[], const unsigned int& len) {
    T tmp;
    for(unsigned int i=0; i<len-1; i++)
        for(unsigned int j=0; j<len-1-i; j++)
            if (a[j] > a[j+1]) // type T must support "operator>"
                tmp = a[j], a[j] = a[j+1], a[j+1] = tmp;
}

int main() {
    const unsigned int len = 5;
    int a[len] = {1,4,3,2,5};
    float f[len] = {3.14,2.72,2.54,1.62,1.41};

    sort(a,len), sort(f,len);
    for(unsigned int i=0; i<len; i++)
        std::cout << a[i] << "\t" << f[i] << std::endl;
}
Overloading templated functions

- Templated functions can be overloaded with templated and non-templated functions
- Resolving an overloaded function call uses the “most specialised” function call
- If this is ambiguous, then an error is given, and the programmer must fix by:
  - being explicit with template parameters (e.g. `sort<int>(...)`)
  - re-writing definitions of overloaded functions
- Overloading templated functions enables meta-programming:
Meta-programming example

```cpp
#include <iostream>

template<unsigned int N> inline long long int fact() {
    return N*fact<N-1>();
}

template<> inline long long int fact<0>() {
    return 1;
}

int main() {
    std::cout << fact<20>() << std::endl;
}
```
Exercises

1. Provide an implementation for:
   \[\text{template<class } T\text{> } T \text{ Stack}<T>::\text{pop}(); \text{ and}\]
   \[\text{template<class } T\text{> } \text{Stack}<T>::\text{~Stack}();\]

2. Provide an implementation for:
   \[\text{Stack(const Stack& s); \text{ and}}\]
   \[\text{Stack& operator=(const Stack& s);}\]

3. Using meta programming, write a templated class \texttt{prime}, which evaluates whether a literal integer constant (e.g. 7) is prime or not at compile time.

4. How can you be sure that your implementation of class \texttt{prime} has been evaluated at compile time?
C and C++

8. The Standard Template Library

Andrew W. Moore

University of Cambridge
(with thanks to Alastair R. Beresford)

Lent Term 2009
The STL

Alexander Stepanov, designer of the Standard Template Library says:

“STL was designed with four fundamental ideas in mind:

- Abstractness
- Efficiency
- Von Neumann computational model
- Value semantics”

It’s an example of generic programming; in other words reusable or “widely adaptable, but still efficient” code
Additional references


Advantages of generic programming

- Traditional container libraries place algorithms as member functions of classes
  - Consider, for example, "test".substring(1,2); in Java
- So if you have $m$ container types and $n$ algorithms, that’s $nm$ pieces of code to write, test and document
- Also, a programmer may have to copy values between container types to execute an algorithm
- The STL does not make algorithms member functions of classes, but uses meta programming to allow programmers to link containers and algorithms in a more flexible way
- This means the library writer only has to produce $n + m$ pieces of code
- The STL, unsurprisingly, uses templates to do this
Plugging together storage and algorithms

Basic idea:

- define useful data storage components, called *containers*, to store a set of objects
- define a generic set of access methods, called *iterators*, to manipulate the values stored in containers of any type
- define a set of *algorithms* which use containers for storage, but only access data held in them through iterators

The time and space complexity of containers and algorithms is specified in the STL standard
A simple example

```cpp
#include <iostream>
#include <vector>  // vector<T> template
#include <numeric>  // required for accumulate

int main() {
    int i[] = {1, 2, 3, 4, 5};
    std::vector<int> vi(&i[0], &i[5]);

    std::vector<int>::iterator viter;

    for (viter = vi.begin(); viter < vi.end(); ++viter)
        std::cout << *viter << std::endl;

    std::cout << accumulate(vi.begin(), vi.end(), 0) << std::endl;
}
```
Containers

- The STL uses *containers* to store collections of objects
- Each container allows the programmer to store multiple objects of the same type
- Containers differ in a variety of ways:
  - memory efficiency
  - access time to arbitrary elements
  - arbitrary insertion cost
  - append and prepend cost
  - deletion cost
  - ...
Containers

- Container examples for storing sequences:
  - `vector<T>`
  - `deque<T>`
  - `list<T>`

- Container examples for storing associations:
  - `set<Key>`
  - `multiset<Key>`
  - `map<Key,T>`
  - `multimap<Key, T>`
Using containers

```cpp
#include <string>
#include <map>
#include <iostream>

int main() {

    std::map<std::string, std::pair<int, int>> born_award;

    born_award["Perlis"] = std::pair<int, int>(1922, 1966);
    born_award["Wilkes"] = std::pair<int, int>(1913, 1967);
    born_award["Hamming"] = std::pair<int, int>(1915, 1968);
    // Turing Award winners (from Wikipedia)

    std::cout << born_award["Wilkes"].first << std::endl;

    return 0;
}
```
std::string

- Built-in arrays and the std::string hold elements and can be considered as containers in most cases
- You can’t call `.begin()` on an array however!
- Strings are designed to interact well with C char arrays
- String assignments, like containers, have value semantics:

```cpp
#include <iostream>
#include <string>

int main() {
    char s[] = "A string ";
    std::string str1 = s, str2 = str1;

    str1[0] = ’a’, str2[0] = ’B’;
    std::cout << s << str1 << str2 << std::endl;
    return 0;
}
```
Iterators

- Containers support *iterators*, which allow access to values stored in a container
- Iterators have similar semantics to pointers
  - A compiler may represent an iterator as a pointer at run-time
- There are a number of different types of iterator
- Each container supports a subset of possible iterator operations
- Containers have a concept of a `beginning` and `end`
## Iterator types

<table>
<thead>
<tr>
<th>Iterator type</th>
<th>Supported operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>== ! = ++ *(read only)</td>
</tr>
<tr>
<td>Output</td>
<td>== ! = ++ *(write only)</td>
</tr>
<tr>
<td>Forward</td>
<td>== ! = ++ *</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>== ! = ++ * --</td>
</tr>
<tr>
<td>Random Access</td>
<td>== ! = ++ * -- + - += -= &lt; &gt; &lt;= &gt;=</td>
</tr>
</tbody>
</table>

- Notice that, with the exception of input and output iterators, the relationship is hierarchical.
- Whilst iterators are organised logically in a hierarchy, they do not do so formally through inheritance!
- There are also const iterators which prohibit writing to ref’d objects.
Adaptors

- An adaptor modifies the interface of another component
- For example the `reverse_iterator` modifies the behaviour of an iterator

```cpp
#include <vector>
#include <iostream>

int main() {
    int i[] = {1, 3, 2, 2, 3, 5};
    std::vector<int> v(&i[0], &i[6]);

    for (std::vector<int>::reverse_iterator i = v.rbegin();
         i != v.rend(); ++i)
        std::cout << *i << std::endl;

    return 0;
}
```
Generic algorithms

- Generic algorithms make use of iterators to access data in a container.
- This means an algorithm need only be written once, yet it can function on containers of many different types.
- When implementing an algorithm, the library writer tries to use the most restrictive form of iterator, where practical.
- Some algorithms (e.g. `sort`) cannot be written efficiently using anything other than random access iterators.
- Other algorithms (e.g. `find`) can be written efficiently using only input iterators.
- Lesson: use common sense when deciding what types of iterator to support.
- Lesson: if a container type doesn’t support the algorithm you want, you are probably using the wrong container type!
Algorithm example

- Algorithms usually take a `start` and `finish` iterator and assume the valid range is `start` to `finish-1`; if this isn’t true the result is undefined

Here is an example routine `search` to find the first element of a storage container which contains the value `element`:

```cpp
//search: similar to std::find
template<class I,class T> I search(I start, I finish, T element)
{
    while (*start != element && start != finish)
    {
        ++start;
    }
    return start;
}
```
Algorithm example

```cpp
#include "example23.hh"

#include "example23a.cc"

int main() {
    char s[] = "The quick brown fox jumps over the lazy dog";
    std::cout << search(&s[0], &s[strlen(s)], 'd') << std::endl;

    int i[] = {1,2,3,4,5};
    std::vector<int> v(i[0], i[5]);
    std::cout << search(v.begin(), v.end(), 3) - v.begin() << std::endl;

    std::list<int> l(i[0], i[5]);
    std::cout << (search(l.begin(), l.end(), 4) != l.end()) << std::endl;
    return 0;
}
```
Heterogeneity of iterators

```cpp
#include "example24.hh"

int main() {
    char one[] = {1,2,3,4,5};
    int two[] = {0,2,4,6,8};
    std::list<int> l (&two[0], &two[5]);
    std::deque<long> d(10);

    std::merge(&one[0], &one[5], l.begin(), l.end(), d.begin());

    for(std::deque<long>::iterator i = d.begin(); i != d.end(); ++i)
        std::cout << *i << " ";
    std::cout << std::endl;

    return 0;
}
```
Function objects

- C++ allows the function call "()" to be overloaded
- This is useful if we want to pass functions as parameters in the STL
- More flexible than function pointers, since we can store per-instance object state inside the function

Example:

```cpp
struct binaccum {
    int operator()(int x, int y) const { return 2*x + y; }
};
```
Higher-order functions in C++

- In ML we can write: `foldl (fn (y,x) => 2*x+y) 0 [1,1,0];`
- Or in Python: `reduce(lambda x,y: 2*x+y, [1,1,0])`
- Or in C++:

```cpp
#include <iostream>
#include <numeric>
#include <vector>

#include "example27a.cc"

int main() { //equivalent to foldl

    bool binary[] = {true,true,false};
    std::cout << std::accumulate(&binary[0],&binary[3],0,binaccum())
              << std::endl; //output: 6

    return 0;
}
```
Higher-order functions in C++

- By using reverse iterators, we can also get foldr:

```cpp
#include <iostream>
#include <numeric>
#include <vector>

#include "example27a.cc"

int main() { // equivalent to foldr

    bool binary[] = {true, true, false};
    std::vector<bool> v(&binary[0], &binary[3]);

    std::cout << std::accumulate(v.rbegin(), v.rend(), 0, binaccum());
    std::cout << std::endl; // output: 3

    return 0;
}
```