C++ Development

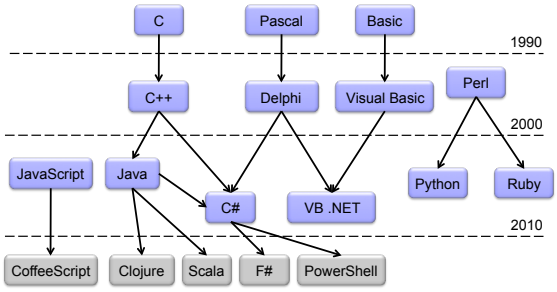
Queens University
March 2013

garth@ggilmour.com

Overview

- This is a 4 day course
  - Core hours and breaks are flexible
- The course has four goals
  - Cover all aspects of the C++ language
  - Use C++ for Object Oriented Development
  - Use C++ templates for Generic Programming
  - Review new features in C++11 (previously known as C++0x)
- Please control the course
  - Ask as many questions as possible
  - Speed up or slow down the pace
  - Request extra examples and exercises
  - Don't sit in misery!!

A Rough Guide to Coding in Industry



The Early History of C++

- 1980-1985 Bjarne Stroustrup adds OO support to C
- 1988 Attempts to standardise C++ begin
- 1990 Annotated C++ Reference Manual published
- 1991 Templates introduced
- 1993 Exception handling introduced
- 1994 The STL becomes an official part of C++
- 1997 - 2001 C++ Standard ratified and debugged

Kinds of C++ Developers

- There are many different kinds of C++ developers
  - Those who transitioned from C
    - Tend to regard the language as 'C with optional objects'
  - Those who learnt C++ in an object oriented manner
    - But before the language was standardised
  - Those who rely on third party libraries
    - Like Roguewave and Microsoft Foundation Classes (MFC)
  - Those who learnt standard C++ as introduced in 1999
    - Including namespaces, exceptions and the STL
  - Those who initially went straight to Java
    - Excellent OO skills but nervous around pointers etc…
  - Those who are using 'Managed C++' (aka C++/CLI)
    - A .NET variant of C++ which has recently been redesigned

Project No 1

Creating a Math Library

garth@ggilmour.com

Creating a Math Library

- We will do the first project together
  - It covers material introduced in the first three chapters
    - The layout of a 'C/C++' program, syntax and the Make tool
  - This will help us kick-start the learning process
- The following artifacts will be produced
  - A file that contains code calling math functions
  - A file that contains the declarations of the math functions
  - A file that contains the definitions of the math functions
  - A file that automatically builds the project

Compiling Code

Creating and Building Simple C and C++ Programs

garth@ggilmour.com

Starting a C/C++ Program

- A C or C++ compiler looks for a function called 'main' and makes it the entry point for the program
  - The program ends when the main function returns
- There are two ways of writing main
  - As a function taking no parameters and returning an integer
    - The integer is an exit status code for the launching application
    - This version is often written as returning nothing (void) which is incorrect, although supported by most older compilers
  - As a function taking two parameters and returning an integer
    - This is used where arguments were passed to the program
    - The first parameter is the number of arguments
    - The second is a data structure that contains the arguments

Starting a C/C++ Program

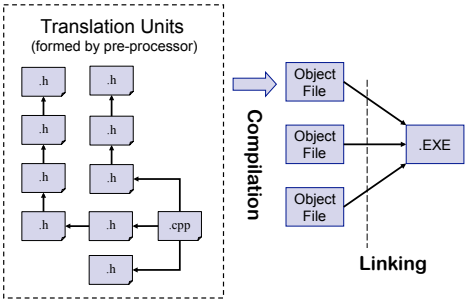
```
//main with no arguments
int main() {
    //program starts here
}

//equivalent to the above
int main(void) {
    //program starts here
}

//an INCORRECT main declaration
void main() {
    //program starts here
}
```

```
//a main method taking arguments
int main(int argc, char * argv [ ]) {
    //program starts here
}

//equivalent to the above
int main(int argc, char ** argv) {
    //program starts here
}
```


Compiling C/C++ Programs

- C and C++ programs are split up
  - Header files contain the declarations of functions and types
  - Source files contain implementations of the functions and types
  - Your program is make up of pairs of '.h' and '.cpp / .c' files
- When source file A uses types from source file B it includes a copy of the header file for B
  - This gives it access to the declarations it needs to compile
- Each source file is compiled into a Translation Unit
  - The visible result of this are object files (nothing to do with OO)
- The linker merges the object files into a finished program
  - It links symbols imported from header files to their definitions

The 3 Stage Compilation Process



Summary of Compiling C++ Code

- The pre-processor forms the Translation Unit
  - By creating an in-memory file
    - Made up of a single code file and zero or more header files
  - Directives allow us to control which code appears in the TU
- The compiler processes each TU in isolation
  - For each Translation Unit an object file is produced
  - These contain 'holes' corresponding to invocations of functions which are declared but not defined in the current TU
- The linker merges the TU's together
  - For each invocation there must be a matching definition
  - Note the definition may be in a library

Declarations Vs. Definitions

- This is critical in understanding C/C++
  - We will be returning to it repeatedly
- In order to compile a TU the compiler needs declarations
  - Promises about what can be found in other TU's
- The most common declaration is a function prototype
  - E.g. 'void func(int i, double d, char * str, void (*ptr)(void));'
- An object file therefore contains 'holes'
  - Function calls that must be wired into the matching definitions
- The linker wires invocations into definitions
  - The definitions may be found in other TU's or in libraries

Linking and Global Variables

- A variable declared outside of any function is 'global'
  - Storage is created at the start of the program
  - The variable is initialized and lives till the program ends
- Code in any function can use the global variable
  - As long as the symbol is visible via a declaration
- Defining the variable twice causes a linker error
  - Placing 'int myvar = 12;' in two TU's fails because it attempts to create two global variables with the same name
- The 'extern' keyword is used to create a declaration
  - E.g. 'extern int myvar = 12;'
  - Function prototypes are implicitly 'extern'

The Preprocessor

- Before code is compiled it is parsed by the preprocessor
  - You can include or exclude symbols and code from the TU
- The preprocessor looks for directives
  - Each directive begins with the '#' symbol
- The '#define' directive defines symbols
  - For example '#define CACHE_SIZE 256'
  - Each instance of 'CACHE_SIZE' will be replaced by '256'
- '#define' can be used to create macros
  - For example '#define MAX(a,b) ((a) > (b) ? (a) : (b))'
  - Macros are now discouraged because they have no type safety
    - Although they are heavily used in libraries like the MFC

Preprocessor Directives

- Symbols and macros can be undefined
  - By using the '#undef' directive
- Code can be conditionally included via if statements
  - '#if', '#ifdef' and 'ifndef' provide the initial condition
    - '#ifdef' and '#ifndef' test for a symbol introduced via a '#define'
  - '#elif', '#else' and '#endif' complete the conditional
- The '#include' directive includes code from a header file
  - The header location can be specified in quotes or angle braces
  - The former is usually a file path while the latter specifies a standard header file supplied with the compiler

Header Guards

- We must ensure that a header file is only included once
  - Or else the compiler sees several copies of any definitions
- This is accomplished via header guards
  - The first time a header is processed we define a symbol
    - Usually a capitalised version of the header file name
  - An '#ifndef' directive ensures the header is not processed again
- The header file may still be parsed
  - If so the preprocessor wastes time finding the '#endif'
  - This can be avoided via redundant header guards
    - Placed around each '#include' in the file doing the including

Header Guards

employee.h

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
        Contents of header file
        only included once
#endif
```

main.c / main.cpp

```
#ifndef EMPLOYEE_H
#include "employee.h"
#endif

// Redundant reader guard prevents
//    the preprocessor consuming all of
//    employee.h in search of the #endif
// Not needed with newer compilers
```


Other Preprocessor Directives

- Other preprocessor directives are rarely used
  - The '#line' directive alters line numbering
  - The '#error' directive terminates compilation
  - The '#pragma' directive is a instruction to the compiler
    - Each compiler has its own set of instructions
    - For example not to emit warnings for a particular kludge
- Some symbols are already defined for your use
  - '__LINE__' and '__FILE__' provide the current line number and name of the file being compiled
  - '__DATE__' and '__TIME__' hold the current time

Preprocessor Directives

Directive	Description
#define	Define a symbol or macro
#undef	Undefine a symbol
#if #elif #else #endif	Conditionally pass text to the compiler
#ifdef #ifndef	Same as #if - but use whether a symbol is (un)defined as the test
#include	Insert a copy of the specified file in the Translation Unit
#pragma	Pass a command to the compiler
#error	Cause a compile time error

Standard I/O

Reading and Writing from the Console, Strings and Files

garth@ggilmour.com

Standard I/O in C/C++

- C/C++ both standardize three kinds of I/O
  - Reading and writing from the console (shell)
  - Storing information in an in-memory buffer
  - Interacting with the file system
- In C code this is done via library functions
  - E.g. 'printf("welcome to the course\n");'
  - E.g. 'gets(delegateName);'
- In C++ code this is done via streams
  - E.g 'cout << "welcome to the course" << endl;
  - E.g 'cin >> delegateName';

Advanced Types of I/O

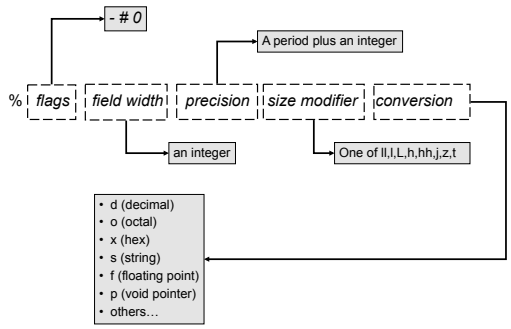
- A major weakness of C/C++ is the lack of platform independent libraries for core tasks
  - Such as network access, regex and encryption
- In both languages you are reliant on OS libraries
  - 'pthreads' for UNIX threads but Win32 threads on Windows
  - 'Motif' for UNIX GUI's but MFC on Windows
  - Networking libraries vary widely
- There are some platform independent C++ libraries
  - Roguewave libraries like 'threads++'
  - The 'QT' GUI library from TrollTech

Writing Output in C

- Console output in C is based around 'printf'
  - We also use 'fprintf' for files and 'sprintf' for buffers
- We pass in a variable number of parameters
  - The first is a string containing 'conversion specifications'
  - The remaining parameters specify the data to be inserted
- For conversion specifications the '%' is followed by:
  - One or more flags to modify the conversion (optional)
  - The width of the field that will be printed (optional)
  - The precision used to print a floating point number (optional)
  - The character that represents what type of data this is


```
#include<stdio.h>

int main() {
	printf("A demo of the different printf options...");
	int i = 27;
	char * msg = "Value of i is ";
	printf("\t%s: %d in decimal\n", msg, i);
	printf("\t%s: %o in octal\n", msg, i);
	printf("\t%s: %x in hexadecimal\n", msg, i);
	printf("\t%s: %6d in decimal padded to 6 places with spaces\n", msg, i);
	printf("\t%s: %06d in decimal padded to 6 places with zeroes\n", msg, i);
	printf("\t%s: %-6d in decimal padded to 6 places and left justified\n", msg, i);
}
```

```
A demo of the different printf options...
	Value of i is : 27 in decimal
	Value of i is : 33 in octal
	Value of i is : 1b in hexadecimal
	Value of i is :     27 in decimal padded to 6 places with spaces
	Value of i is : 000027 in decimal padded to 6 places with zeroes
	Value of i is : 27     in decimal padded to 6 places and left justified
```

Reading Input in C

- Console output in C is based around 'scanf'
  - We also use 'fscanf' for files and 'sscanf' for buffers
- Once again we use a variable number of parameters
  - The first is a string that controls how the input is formatted
    - This string contains 'conversion specifications' as used in 'printf'
  - The remaining parameters specify where the data is to be stored
    - Note the C/C++ problem of 'magic numbers' for buffer sizes
- Items not matching the specification are ignored
  - For this reason it is safer to consume an entire line of input with 'gets' and then parse it as many times as needed with 'sscanf'


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

int main() {
	printf("A demo of the different scanf options…\n");

	char name[100];
	char dept[100];
	int age;
	float salary;

	strcpy(name, "nobody");
	strcpy(dept, "nowhere");
	age = salary = 0;

	printf("Please enter '<name> <dept> <age> <salary>'\n");
	scanf("%s %s %d %f", name, dept, &age, &salary);
	printf("You are %s of age %d working in %s for %.2f\n",name, age, dept, salary);
}
```


Choosing a Language for I/O

- C/C++ are great for binary I/O
  - Especially when performance is critical and platform specific API's are being used to read/write
- For other tasks they are not so good
  - Beware the 'everything looks like a nail' mindset
- Use the best language for the job
  - For text processing use Perl
    - Or one of its competitors like Python or Ruby
  - For networking use Java
    - It has the most comprehensive and friendly networking library
    - It's also probably the best language for working with XML

The ‘scanf’ Demo - Perl Version

```
print "A demo of the different scanf options...\n",
      "Please enter 'Name=<name> Dept=<dept> Age=<age> Salary=<salary>'\n",
      "\t Options can be entered in any order\n";

$line = <STDIN>;
@tokens = split /\s+/, $line;
foreach(@tokens) {
        if($_ =~ m/(\w+)=(\w+)/) {
                $params{lc($1)} = $2;
        }
}
print "You are $params{'name'} of age $params{'age'} ",
        "working in $params{'dept'} for $params{'salary'}\n";
```

```
A demo of the different scanf options...
Please enter 'Name=<name> Dept=<dept> Age=<age> Salary=<salary>'
        Options can be entered in any order
dept=it nAme=dave sALAry=30000 age=27
You are dave of age 27 working in it for 30000
```


Input and Output in C++

- I/O in C++ uses streams rather than functions
  - A stream is a sequence of bytes or characters being read from a source or sent to a destination
  - You use the ‘<<’ and ‘>>’ operators to send and receive
- There are three built in streams for you to use
  - The ‘cin’, ‘cout’ and ‘cerr’ streams are tied to the console
  - To write to the console use ‘cout << “2 + 2 is: ” << 4 << endl’;
- Note that modern C++ libraries live in namespaces
  - We reference types in another namespace via ‘using’
  - The standard libraries use the namespace std

Input and Output in C++

```
//include the stream classes
#include<iostream>

//import the contents of the standard
// library into our program
using namespace std;

//the entry point
int main() {
      //write a message followed by
      // the end of line character
      cout << "hello world" << endl;

      //explicitly provide an exit code
      return 1;
}
```

Input and Output in C++

- Building formatted strings in C is awkward
  - It can be done using functions like 'itoa' and 'sprintf'
- C++ provides string streams as an alternative
  - A string stream works the same way as 'cin' and 'cout' except that the source and/or sink is a 'std::string'
  - This makes it easy to create or parse text
- There are three kinds of string stream
  - An 'ostringstream' can only be written to
  - An 'istringstream' can only be read from
  - A 'stringstream' supports both input and output


```
#include <iostream>
#include <sstream>

using namespace std;

int main() {
	ostringstream output;
	output << "Values are: ";
	output << 12;
	output << " ";
	output << 34.56;
	output << " ";
	output << false;
	output << endl;

	string message = output.str();
	cout << message;
}
```

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

int main() {
	istringstream input("abcdef 12 3.45 true");

	string sval;
	int ival;
	double dval;
	bool bval;

	input >> sval;
	input >> ival;
	input >> dval;
	input >> bval;

	if(bval) {
		cout << "Total is: " << ival + dval << endl;
	} else {
		cout << "Cannot read from stream" << endl;
	}
}
```


Types & Operators

The Basics of Programming

garth@ggilmour.com

Comments and Conventions

- C and C++ both support two styles of comment
- The '//' comment is for single lines
  - The compiler ignores everything till the end of the current line
  - Note this was not available in C till 'C99'
- The '/* … */' comment is multi-line
  - The '/*' token causes the compiler to ignore everything that follows until a '*/' token is found
  - This style of comment cannot be nested
- You can use the preprocessor for commenting
  - '#if 0' stops code reaching the compiler till the matching '#endif'

Numeric Data Types

- C supports a range of numeric data types
  - Char, short, int, long and long long for integers
  - Float, double and long double for floating point numbers
- The numeric types can be signed or unsigned
- The '_Bool' type was introduced in C99
  - It is an unsigned integer that only holds '0' or '1'
  - The type was not called 'bool' for backward compatability
- The size of a data types can vary between platforms
  - To remain portable your code shouldn't make assumptions

Numeric Data Types in C

Data Type	Description
char	An ASCII character
short	A small integer
int	An integer
long	A large integer
long long	A very large integer
float	A floating point number
double	A large floating point number
long double	A very large floating point number
_Bool	A boolean value

Enumerations

- An enumeration is a user-defined data type
  - Made up of a list of constant values
    - E.g. 'enum Colors { RED, GREEN, BLUE };'
    - E.g. 'enum Comms { OPEN, CLOSED, SENDING } state;'
- Enumerations are integers in disguise
  - The first constant is assigned the value 0 and so on
  - Using them as integers is not a good idea
- Values can be explicitly assigned to constants
  - E.g. 'enum Payments { CREDIT_CARD = 20, CASH = 30 }'
  - If CASH had not been given an explicit value it would be 21

Unions

- A union is an aggregate type
  - It is made up of fields of different types
  - All the fields share the same storage space
- The size of union is the size of the largest field
  - Plus any padding required by the OS memory model
  - Unions are useful when you need to declare a number of large variables, but only one of them will be in use at a time

```
union test {
      short f1[4];
      double f2;
      char * f3;
}
union test myvar.f2 = 12.3;
myvar.f3 = "abcd";
```


Literal Values

- A literal value is a constant used to initialize a variable
  - Literals should not be used for 'magic numbers'
- An integer literal is just a number e.g. 18
  - If the number is prefixed by 0 it is octal e.g. 018
  - If the number if prefixed by 0x it is hex e.g. 0x18
- Integer literals are of type int by default
  - Literals of type long are followed with 'l' or 'L'
- A floating point literal has a decimal point e.g. 3.4
  - The literal is of type double by default, follow it with 'F' or 'f' to specify float or alternatively 'l' or 'L' to specify long double
  - Use 'E' or 'e' if you want to use scientific notation e.g. 3.4E-5

Literal Values

- Character literals are letters in single quotes
  - A character literal preceded by 'L' is of type 'wchar_t'
  - C/C++ defines special characters called escape sequences
    - These represent formatting codes used in documents
    - They are written as a backslash and then a character
- A sequence of chars in double quotes is a string literal
  - This is a array of characters terminated by the null char '\0'
  - So "abcdef" is the characters 'a' 'b' 'c' 'd' 'e' 'f' and '\0'
  - As with character literals the string may be prefixed with 'L'

Literal Values

- Note that on Windows a new line is '\r' followed by '\n' (often called CRLF) whereas UNIX uses just '\n'

Escape Sequence	Meaning
\n	A new line
\r	A carriage return
\t	A horizontal tab
\v	A vertical tab
\\	A backslash
\'	A single quote
\"	A double quote
\b	A backspace

Declaring and Defining Variables

- A variable declaration is a type followed by a name followed optionally by an initial value
  - For example 'double d;', 'int i = 27;' or 'char c = 'A';'
- A local variable should be initialized before it is used
  - Otherwise its value is undefined, and (even if compiled) calculated based on the bit pattern originally in the memory
  - The initializer can be a literal, expression or return value
- Global variables are automatically initialized to zero
  - But global data should be avoided as it is too open to abuse

Casting and Type Conversions

- One type can be converted to another via a cast
  - E.g. 'int myint = (int)myfloat;'
- Casting performs an explicit conversion
  - This will always be preformed without an error
  - However the value is the receiver may be mangled
- The compiler also performs implicit conversions
  - E.g. assignment, arithmetic with mismatched types or calling a function without an exact match to the parameter(s)
  - The set of implicit conversions is smaller than for explicit ones
    - Different compiler may allow different conversions

Converting To and From Strings

- There is no built in concatenation operator
  - If you want to convert the string "123" to the integer 123 you must do this via a function call
- To convert a string to a number:
  - The functions 'atoi', 'atol' and 'atof' have traditionally been used
  - In modern C code the more powerful functions 'strtod', 'strtol' and 'strtof' should be used instead
- The 'sprintf' function is also very useful
  - You can use it to combine concatentaion and formatting
  - E.g. 'sprintf(buffer, "%f purchased at %d percent discount", f, i);'

C Operators

- C supports standard numeric and logical operators
  - '+', '-', '*', '/' and '%' for arithmetic
  - '==', '>', '<', '<=' and '>=' for comparison
  - '&&' and '||' for logical AND and OR
- All the numeric operators have a 'n=' form
  - For example 'i += 7' is equivalent to ' i = i + 7'
- The '--' and '++' operators increment or decrement by 1
  - If placed before the variable it is changed and then used
    - So after 'y = 5; x = --y;' both x and y are equal to 4
  - If placed after the variable it is used and then changed
    - So after 'y = 5; x = y--;' the values are 5 in x and 4 in y

C Operators

Arithmetic Operators	
* , *=	multiply
/ , /=	divide
% , %=	modulus
+ , +=	addition
- , -=	subtraction
++	prefix and postfix increment
--	prefix and postfix decrement
Boolean Operators	
! , && , | |	Logical NOT, AND and OR
&, | , ^	Bitwise AND, OR and XOR
== , !=	equality and inequality
< , <=, > , >=	comparisons

The ‘sizeof’ Operator

- ‘sizeof’ returns the size of a type
  - It can be passed either a type name or variable
    - E.g. ‘sizeof(long)’, ‘sizeof(myvar)’ or ‘sizeof(struct S)’
  - Any padding is not counted as part of the size
    - E.g. space added to align fields of a type for efficient access
- The operator has a few quirks:
  - You can use ‘sizeof’ on array variables
    - In which case it returns the size of the entire array
    - E.g. if ‘sizeof(int)’ returns 4 and then given the declaration ‘‘int myarray[4]’ the expression ‘sizeof(myarray)’ returns 16
  - You can use ‘sizeof’ on expressions:
    - If ‘s1’ and ‘s2’ are of type short then ‘sizeof(s1 + s2)’ returns 4

Miscellaneous Operators

- The conditional operator is an alternative to ‘if’
  - It is used to conditionally initialise a variable
    - E.g. ‘i = a ? 7 : 6;’ is equivalent to ‘if(a) { i = 7; } else { i = 6}’
  - This is the only ternary (three operand) operator in ‘C’
- The ‘!’ operator acts as a logical NOT
  - A true statement becomes false and vice versa
- Some operators treat integer types as lists of bits
  - ‘$’, ‘|’ and ‘^’ represent bitwise AND, OR and NOT
  - ‘<<’ and ‘>>’ bit-shift the contents of a variable

Control Structures

Conditionals and Loops

garth@ggilmour.com

The IF Conditional

- An 'if' statement is the most common conditional
  - The syntax is 'if(*condition*) { … } else { … }'
  - The condition will be converted to a boolean value
- If statements can be chained together
  - E.g. ''if(*conditionA*) { … } else if(*conditionB*) { … }''
  - There is no concept of 'elseif' or 'elif'
- Braces can be omitted if the body has a single statement
  - 'if(*condition*) a++; b++;' increments 'a' if *condition* is true
    - The variable 'b' will always be incremented regardless
  - It is NOT a good idea as it harms maintainability


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

int main() {
	printf("Enter a number…\n");

	int number;
	scanf("%d", &number);

	if(number < 10) {
		printf("Number is less than ten\n");
	} else if(number < 20) {
		printf("Number is less than twenty\n");
	} else if(number < 30) {
		printf("Number is less than thirty\n");
	}else {
		printf("Number is thirty or greater\n");
	}
}
```

The Switch Condition

- A switch tests if a number belongs to a set of values
  - The switch block encloses a number of case statements
    - Each of which has an associated value
  - The number must be an integer and case values must be unique
- An optional 'default' statement is triggered if none match
  - Most developers consider it a best practise to put this in
- Each case section must be terminated by a 'break'
  - If it isn't the code 'falls through' and executes the next case
  - If this is what you want you should always add a comment


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

int main() {
	printf("Enter a number…\n");
	int number;
	scanf("%d", &number);

	switch(number) {
	case 10:
		printf("Number is 10\n");
		break;
	case 20:
		printf("Number is 20\n");
		break;
	case 30:
	case 40:
	case 50:
		printf("Number is 30, 40 or 50\n");
		break;
	default:
		printf("Unknown number %d\n", number);
	}
}
```



```
printf("Type of HTTP request is: ");

switch(getRequestType()) {
	case GET:
		printf("GET");
		break;
	case POST:
		printf("POST");
		break;
	case HEAD:
		printf("HEAD");
		break;
	case PUT:
		printf("PUT");
		break;
	default:
		printf("UNKNOWN");
}
```

```
printf("Type of HTTP request is: ");

switch(getRequestType()) {
	case GET:
	case HEAD:
		printf("WITHOUT BODY");
		break;
	case POST:
	case PUT:
		printf("WITH BODY");
		break;
	default:
		printf("UNKNOWN");
}
```

Optimizing Conditionals

- There are ways of speeding up conditionals
  - Which do not harm the readability of your code
- Take advantage of scoped variable declarations
  - Don't declare all your variables at the start of the function
  - Instead limit them to the narrowest scope possible
  - This lets us benefit from 'lazy evaluation'
    - We only create and initialize variables when necessary
- Take advantage of the 'short-circuit' effect
  - Given 'if(foo() || bar())' place the call with the highest probability on the left hand side of the expression

Iteration in C/C++

- C/C++ have three different loops
  - The 'while' loop
  - The 'do … while' loop
  - The 'for' loop
- There is no built in construct for an infinite loop
  - But it is easy to create one
- As with an 'if' statement braces can be omitted
  - If the body only contains a single statement
  - Once again this is not good practise
    - It is very bad for code maintenance

The While Loop

- The while loop is the most basic
  - Its syntax is 'while(*condition*) { … }'
  - It will execute zero or more times
- 'while(1) {…}' creates an infinite loop
  - Note that 'for(; ;) { … }' is traditionally preferred
  - As there is no danger of the compiler rechecking '1'
- A do-while loop executes at least once
  - The syntax is 'do { … } while(*condition*);'
  - Note the semicolon is required at the end

```
int main() {
	printf("Enter a number…\n");
	int number;
	scanf("%d", &number);
	int countOne = 0;
	while(countOne < number) {
		printf("While loop message: %d\n", countOne++);
	}
	printf("--------------------\n");

	int countTwo = 0;
	do {
		printf("Do While loop message: %d\n", countTwo++);
	} while(countTwo < number);
	printf("--------------------\n");

	int countThree = 0;
	while(1) {
		if(countThree == number) {
			break;
		} else {
			printf("Infinite While loop message: %d\n", countThree);
		}
		countThree++;
	}
}
```


The For Loop

- The ‘for’ loop is more complex than the others
  - It is used to loop a specific number of times
  - It is bad practice to break out of a ‘for’ loop
- Its declaration consists of three parts
  - The initialization of one or more local variables
  - The condition that will end the loop
  - Actions to be taken after each iteration
- For loops are often used to walk a data structure
  - Typically an array or a vector (a dynamic array)
- Most developers expect the canonical form:
  - E.g. ‘for(int i=0;i<MAX;i++) { … }’


```
int main() {
	printf("Enter a number…\n");
	int number;
	scanf("%d", &number);

	for(int i=0;i<number;i++) {
		printf("First for loop message: %d\n", i);
	}
	printf("--------------------\n");

	for(int i=number;i>0;i--) {
		printf("Second for loop message: %d\n", i);
	}
	printf("--------------------\n");

	int count = 0;
	for(;;) {
		if(count == number) {
			break;
		} else {
			printf("Third for loop message: %d\n", count);
		}
		count++;
		}
}
```

Exiting From a Loop

- There are three ways to terminate a loop
  - Use a 'return' to exit the containing function
  - Use a 'break' to exit the current loop
    - This only breaks you out of the innermost loop
    - Breaks can only occur in a switch or a loop
  - Use a 'goto' to jump to another point in the program
    - The use of goto's is heavily discouraged
- A 'continue' statement ends the current iteration
  - The remainder of the current loop is skipped and execution continues with the re-evaluation of the condition
    - A 'continue' can only occur inside a loop

Picking the Right Type of Loop

- Don't automatically default to a 'while' loop
  - Sometimes a 'do …while' is clearer and more efficient
- Clearly comment unusual 'for' loops
  - But don't be afraid to use different conditions
- Don't duplicate code through use of the wrong loop
  - E.g. calling 'readLine' before and inside a loop
- Don't keep calculating values redundantly
  - If a value will remain the same within all iterations then calculate it before you enter the loop

Which is the Right Loop?

```
char * line = readLine();
while(!isEndToken(line)) {
    processData(line);
    line = readLine();
}
```

```
char * line;
do {
    line = readLine();
    processData(line);
} while((!isEndToken(line));
```

```
while(1) {
    char * line = readLine();
    if(!isEndToken(line)) {
        break;
    }
    processData(line);
}
```

```
for(;;) {
    char * line = readLine();
    if(!isEndToken(line)) {
        break;
    }
    processData(line);
}
```

Compiler Optimizations

- The compiler may optimize your loops and conditionals
  - When it can detect there is a more efficient implementation that is functionally equivalent to the original code
- Consider 'for(int i=0;i<MAX;i++) { op1(); }'
  - If the compiler knows that MAX is always 3 then it can replace the loop with 3 calls to 'op1'
- Some assumptions are incorrect in multithreaded code
  - Consider 'i =12; if(i > 0) { … }' when 'i' is a global variable
  - In single threaded code the conditional can be removed
  - In multithreaded code it is essential it remain
    - This is the purpose of the 'volatile' keyword

Project No 2

Building a Pyramid

garth@ggilmour.com

Printing a Pyramid

- Write a program to print a pyramid
  - Based on a height supplied by the user
  - Hint - the number of spaces is 'height - 1' for the first row

```
____#
___###
__#####
_#######
#########
```

Basic Functions

'C' Style Function Declarations

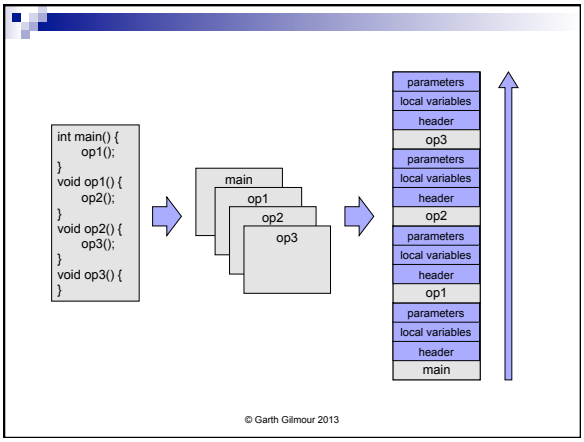
garth@ggilmour.com

Function Declarations

- We have already seen the basics of functions
  - And how they are used in the compile->link process
- A function is divided into two parts
  - The declaration tells the compiler the function exists
    - Its specifies the functions name and signature
  - The definition provides the functions implementation
    - This is typically in another Translation Unit or library
- Compilers don't wire function calls into implementations
  - Instead this is the responsibility of the linker
  - How linkers work is an interesting topic in its own right

Functions and the Call Stack

- Function calls are organized into a Call Stack
  - Every call is represented as a new frame on the stack
  - There may be limits on the size of the stack
    - Either explicit (a magic number defined by the OS) or implicit (defined by the amount of space in the process)
    - Usually these are too large to be relevant
- The contents of a frame include:
  - Memory set aside for parameters and local variables
  - The instruction to return to in the calling function
- The compiler determines the size of each frame
  - I.e. the amount of space required for a call to a function



The Call Stack and Threads

- A multi-threaded program has multiple call stacks
  - When you start a new thread you launch a new call stack
    - Based on an arbitrary function which becomes the 'main'
    - Usually the function must have a particular signature
  - Global and heap memory is shared between threads
- CPU time must be shared out between threads
  - The OS may allocate time to individual threads
  - Alternatively each processes may be given time to share out amongst its threads as it sees fit
- The main thread must be kept alive
  - To prevent the application ending prematurely

Function Prototypes in Depth

- Early versions of C didn't use prototypes
  - This lead to all kinds of hard to debug errors
- In C functions with no parameters should use 'void'
  - E.g. 'int func(void)' or 'void func(void)'
  - The void is optional but its use is recommended
- Functions with parameters must list their types
  - E.g. 'int func(int, float, char *, short)'
  - Naming the types is optional but helps maintainability
- Functions can take a variable number of arguments
  - E.g. 'int func(int , double, …)'

Functions and Recursion

- The call stack structure enables recursion
  - We can have multiple frames for calls to the same function
  - Each frame has its own set of parameters and local variables
- A function can call itself any number of times
  - But there must be a terminating condition that ends the recursion before all the stack space is used up
- Any task that uses a loop can be done with recursion
  - Some solutions are easier to implement via recursion

Functions in Depth

- C/C++ only passes parameters by value
  - A copy is made of the data provided by the caller
  - But that data may be a memory address
    - In which case the parameter is declared as a pointer
  - Parameters may be given the 'register' storage specifier
- Functions may be declared 'inline' or 'static'
  - Inline is a hint to the compiler that the implementation may be inserted directly into the function call
  - Static means the functions name will not be visible to the linker
    - This may result in the compiler making the function inline

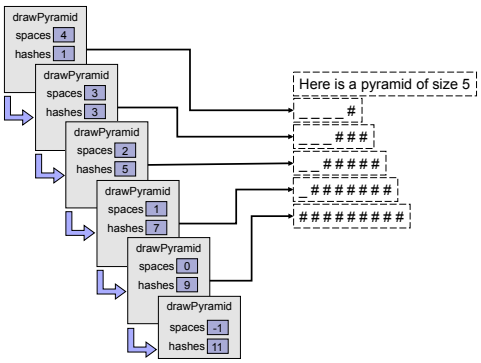
Project No 3

Building a Pyramid Recursively

garth@ggilmour.com

Printing a Pyramid Recursively

- Modify your solution for printing a pyramid
  - By making the implementation recursive
  - The functionality should remain the same
- One possible approach is:
  - Create a 'drawPyramid' function
    - Which prints a single row of the pyramid
    - It takes the number of spaces and hashes
  - The function should call itself
    - Passing in 'spaces - 1' and 'height + 2'
  - You will need a terminating condition
    - Such as when the number of spaces is negative



Structures & Arrays

Data Structures in C/C++

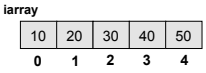
garth@ggilmour.com

Arrays

- The most basic data structure is an array
  - An array is a contiguous set of variables lined up in memory
  - The first box in the array always has index zero
- An array is declared by giving the size in square braces
  - For example 'int iarray[10]' creates an array of ten integers, lined up as boxes in memory and indexed from zero to nine
- An array does not know its own length
  - You need to store this information separately
  - Or else store a special terminator value in the last box

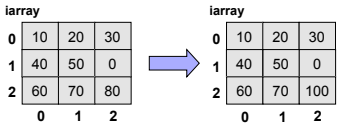
Arrays

- You can specify the contents of an array in the definition
  - For example 'int iarray[ ] = { 10,20,30,40,50 }'
- Arrays can be multidimensional
  - For example 'int iarray[3][4]' creates a 2d grid
- These arrays can also be initialised via braces
  - E.g. 'int iarray [ ][ ] = { {10,20,30,40}, {11,12,13,14}, {6,7,8,9} } '
  - 'int iarray [3][4] = { 10,20,30,40,11,12,13,14,6,7,8,9 } ' would have the same effect but is rarely used



Arrays

```
int iarray [ ][ ] = {
                        {10,20,30},
                        {40,50},
                        {60,70,80}
                    };
iarray [2][2] = 100;
```



Arrays

- Arrays cannot be returned from functions
  - However you can return the address of an array
- Normally the compiler must know the size of the array
  - Otherwise it cannot reserve the memory
    - On the call stack in the case of a local variable
    - In global memory in the case of a global variable
  - Hence the code 'int n = op1(); float myarray[n];' will not compile
    - However arrays can be created from heap memory…
- C99 adds a feature known as 'variable length arrays'
  - This allows the array size to be a value computed at runtime

Structures

- An array is a sequence of a single type
  - This doesn't help us when we want to group a set of different types to describe some abstraction (e.g. an Employee)
- Structures let us create Abstract Data Types
  - We can group a sequence of variables of different types in memory and manipulate them as a unit
  - We can pass them to functions or place them in arrays
- Fields of a structure are accessed using the '.' operator
  - If we say 'e.age' where age is the 2nd field in an Employee struct
  - The compiler knows to go to 'address of e plus size of field 1'

Structures

```
struct Employee {
      int age;
      float salary;
      char name[10];
      char dept[10];
};
```

Age			Salary	
N	a	m	e	
D	e	p	t	

```
struct Employee e;
e.age = 27;
e.salary = 30000;
strcpy(e.name, "Joe Smith");
strcpy(e.dept,"Marketing");
```

27			30000	
J	o	e		S
m	i	t	h	\0
M	a	r	k	e
t	i	n	g	\0

Structures

- Structures can be self-referential
  - This is especially useful in collections
- Instances can be initialized as with arrays
  - Values for fields are given in their order of declaration

```
struct Node {
      double value;
      struct Node * next;
      struct Node * prev;
};
```

```
Node n1 = {12.34, 0, 0};
Node n2 = {56.78, 0, &n1};
n1.next = &n2;
```

```
struct Stuff {
      int val1;
      double val2;
} myarray []  = { {3,4.5} , {6,7.8} };
```


Pointers Part 1

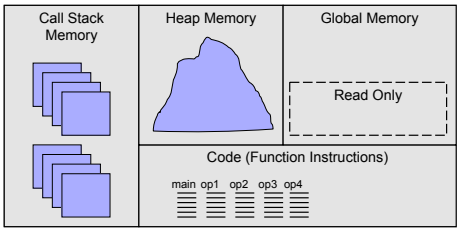
Scalars, and Arrays

garth@ggilmour.com

Memory Management

- Managing memory is a troublesome feature of C/C++
  - Which is why Java and C# adopted a garbage collector
- Data can be allocated in three places
  - Global data is allocated outside any function
    - Initialised before main begins and valid for the programs lifetime
  - Local variables are allocated on the call stack for each function
    - Each function call adds a new frame to the stack
  - Memory can be dynamically allocated from the heap
    - Via ‘malloc’ and ‘free’ in C and ‘new’ and ‘delete’ in C++
    - The developer is responsible for returning the memory to the heap

Addressable Areas of Memory



Using Pointers

- A pointer is a variable that holds the address of another component of your program
  - You can take the address of most things
- '*' is used both to declare and dereference a pointer
  - E.g. the syntax 'int * ptr = func()' initializes ptr to point to the integer returned from function func
  - The syntax 'int val = *ptr' gets the value 'ptr' points to
- All pointers have the same size
  - They must be big enough to hold any possible memory address
  - We give a pointer a type so that when we dereference it the compiler knows how may bits to grab and how to interpret them
  - The void pointer type 'void * v_ptr' can point to anything

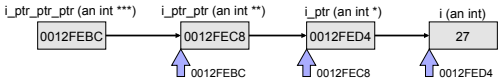
Using Pointers (C Version)

```
#include <stdio.h>
int main() {
	int i = 27;
	int * i_ptr = &i;
	printf("value in i is %d\n", i);
	printf("address of i is %x\n", &i);
	printf("value in pointer is %x\n", i_ptr);
	printf("value from dereferencing pointer is %d\n", *i_ptr);
	int ** i_ptr_ptr = &i_ptr;
	printf("address of i_ptr is %x\n", &i_ptr);
	printf("value in pointer to pointer is %x\n", i_ptr_ptr);
	printf("value from dereferencing pointer to pointer is %x\n", *i_ptr_ptr);
	int *** i_ptr_ptr_ptr = &i_ptr_ptr;
	printf("address of i_ptr_ptr is %x\n", &i_ptr_ptr);
	printf("value in pointer to pointer to pointer is %x\n", i_ptr_ptr_ptr);
	printf("value from dereferencing pointer to pointer to pointer is %x\n", *i_ptr_ptr_ptr);
}
```

Using Pointers (C++ Version)

```
int main() {
	int i = 27;

	int * i_ptr = &i;
	cout << "value in i is " << i << endl;
	cout << "address of i is " << &i << endl;
	cout << "value in pointer is " << i_ptr << endl;
	cout << "value from dereferencing pointer is " << *i_ptr << endl;

	int ** i_ptr_ptr = &i_ptr;
	cout << "address of i_ptr is " << &i_ptr << endl;
	cout << "value in pointer to pointer is " << i_ptr_ptr << endl;
	cout << "value from dereferencing pointer to pointer is " << *i_ptr_ptr << endl;

	int *** i_ptr_ptr_ptr = &i_ptr_ptr;
	cout << "address of i_ptr_ptr is " << &i_ptr_ptr << endl;
	cout << "value in pointer to pointer to pointer is " << i_ptr_ptr_ptr << endl;
	cout << "value from dereferencing pointer to pointer to pointer is " << *i_ptr_ptr_ptr << endl;
}
```


Using Pointers




```
value in i is 27
address of i is 0012FED4
value in pointer is 0012FED4
value from dereferencing pointer is 27
address of i_ptr is 0012FEC8
value in pointer to pointer is 0012FEC8
value from dereferencing pointer to pointer is 0012FED4
address of i_ptr_ptr is 0012FEBC
value in pointer to pointer to pointer is 0012FEBC
value from dereferencing pointer to pointer to pointer is 0012FEC8
```


Using Pointers With Arrays

- The name of an array is its address
  - A pointer can be initialized via the name of an array
    - The pointer holds the address in memory of its first element
    - E.g. 'int * iptr = myarray' is the same as 'int * iptr = &myarray[0]'
- Incrementing a pointer advances it by the size of its type
  - So if an int is 32 bits wide the expression 'i_ptr++' moves 'i_ptr' 32 bits forward rather than just by one
  - This is often used to iterate over an array
    - Or parse a data structure with a well defined format
- The '[]' operator works the same as in arrays
  - So 'i_ptr[3]' returns the integer value 96 bits ahead of 'i_ptr'

Using Pointers With Arrays

- Arrays and pointers are closely associated
  - E.g. you can write 'myarray[2]' or 'i_ptr[2]'
  - But note that a pointer is a variable in its own right

```
void main() {
      int myarray[10] = {101,202,303,404,505,606,707,808,909,0};
      int * i_ptr = myarray;
      while(*i_ptr != 0) {
            cout << *i_ptr << endl;
            i_ptr++;
      }
      int * i_ptr2 = myarray;
      int count = 0;
      while(i_ptr2[count] != 0) {
            cout << i_ptr2[count++] << endl;
      }
}
```


Project No 3

Copying Arrays

garth@ggilmour.com

Copying Arrays

- Try to implement the functions declared below
  - After 'addArrays' has been called 'iarray3' should hold the values 110, 220, 330, 440 and 550

```
#include <stdio.h>

void addArrays(int * ptr1, int * ptr2, int * ptr3, int size);
void print(int * ptr1, int size);

int main() {
      int iarray1[] = {10,20,30,40,50};
      int iarray2[] = {100,200,300,400,500};
      int iarray3[5];

      addArrays(iarray1,iarray2,iarray3, 5);
      print(iarray3,5);
}
```

Pointers Part 2

String Manipulation in C/C++

QUB – March 2013

garth@ggilmour.com

String Manipulation In C

- In C code strings are simply arrays of char
  - Which are terminated by the null char '\0'
    - Not to be confused with the null pointer '0'
- The compiler adds a null char to string literals
  - Make sure the strings you build are null terminated
    - Otherwise your code will continue reading memory
      - Until it finds a null or causes a memory access error
    - All the standard functions rely on the ending null
  - Remember to leave space in char arrays for the null

const char * str = "ABCD";

'A'	'B'	'C'	'D'	'\0'
0	1	2	3	4

String Manipulation In C

Function Signature	Description
char * strcat(char * dest, const char * src)	Appends the chars in 'src' to 'dest'
char * strncat(char * dest, const char * src, size_t n)	Appends 'n' chars from 'src' to 'dest'
int strcmp(const char * s1, const char * s2) int strncmp(const char * s1, const char * s2, size_t n)	Compares all or part of two strings, returning a value greater than, less than or equal to zero
char * strcpy(char * dest, const char * src) char * strncpy(char * dest, const char * src, size_t n)	Copies all or part of one string into another
size_t strlen(const char * s)	Returns the number of chars in the string
char * strchr(const char * s, int c) char * strrchr(const char * s, int c)	Finds the first or last occurrence of a char
size_t strspn(const char * s, const char * set)	Finds the distance to the first char not in 'set'
size_t strcspn(const char * s, const char * set)	Finds the distance to the first char from 'set'
char * strpbrk(const char * s, const char * set)	Finds the first char which occurs in 'set'
char * strstr(char * str, const char * set)	Finds the first occurrence of 'set' in 'str'
char * strtok(char * str, const char * set)	Tokenizes 'str' using the chars in 'set'

String Manipulation In C

- String functions assume the parameters are valid
  - When characters are moved it is assumed that the destination string is large enough to hold them
  - The null char is taken as the end of the string
    - Any following characters in the array are ignored
  - String literals should not be passed for modification
    - The effect of changing a string literal is undefined
- Wide-string versions of the functions are available
  - E.g. wcscat, wcscmp, wcscpy and wcslen
  - These take parameters of type 'wchar_t' and 'wchar_t *'
    - This is a multi-byte wide character for internationalization

Finding the Length of a String

```
void demoStrlen(const char * p1) {
	int length  = strlen(p1);
	cout << "String length is: " << length << endl;
}
int main() {
	const char * str1 = "ab";
	const char * str2 = "abcd";
	const char * str3 = "abcdef";
	const char * str4 = "abcdefgh";
	const char * str5 = "abcdefghij";
	const char * str6 = "abcdefghijkl";
	const char * str7 = "abcdefghijklmn";

	demoStrlen(str1);
	demoStrlen(str2);
	demoStrlen(str3);
	demoStrlen(str4);
	demoStrlen(str5);
	demoStrlen(str6);
	demoStrlen(str7);
}
```

```
String length is: 2
String length is: 4
String length is: 6
String length is: 8
String length is: 10
String length is: 12
String length is: 14
```


Copying One String To Another

```
void demoStrcpy(char * p1, const char * p2) {
	strcpy(p1,p2);
	cout << p1 << endl;
}
int main() {
	const char * str1 = "cdef";
	const char * str2 = "ghij";
	const char * str3 = "klmn";
	const char * str4 = "opqr";
	const char * str5 = "stuv";
	const char * str6 = "wxyz";
	char buffer[] = {'a','b','a','b','\0'};

	cout << buffer << endl;
	demoStrcpy(buffer,str1);
	demoStrcpy(buffer,str2);
	demoStrcpy(buffer,str3);
	demoStrcpy(buffer,str4);
	demoStrcpy(buffer,str5);
	demoStrcpy(buffer,str6);
}
```

```
abab
cdef
ghij
klmn
opqr
stuv
wxyz
```

Concatenating Strings Together

```
void demoStrcat(char * p1, const char * p2) {
	strcat(p1,p2);
	cout << p1 << endl;
}
int main() {
	const char * str1 = "cdef";
	const char * str2 = "ghij";
	const char * str3 = "klmn";

	char buffer[27];
	buffer[0] = 'a';
	buffer[1] = 'b';
	buffer[2] = '\0';

	cout << buffer << endl;

	demoStrcat(buffer,str1);
	demoStrcat(buffer,str2);
	demoStrcat(buffer,str3);
}
```

```
ab
abcdef
abcdefghij
abcdefghijklmn
```


Comparing Strings

```
void demoStrcmp(const char * p1, const char * p2) {
	int retval = strcmp(p1,p2);
	if(retval > 0) {
		cout << "First string is greater" << endl;
	} else if(retval < 0) {
		cout << "Second string is greater" << endl;
	} else {
		cout << "Strings are equal" << endl;
	}
}
int main() {
	const char * str1 = "abcdef";
	const char * str2 = "abcdef";
	const char * str3 = "abcefg";

	demoStrcmp(str1,str2);
	demoStrcmp(str2,str3);
	demoStrcmp(str3,str2);
}
```

```
Strings are equal
Second string is greater
First string is greater
```


Searching For Chars In A String

```
int distance(const char * p1, const char * p2) {
	int count = 0;
	while(p1 != p2) {
		count++;
		p1++;
	}
	return count;
}
void demoStrchr(const char * p1, char p2) {
	char * first_pos = strchr(p1,p2);
	char * last_pos = strrchr(p1,p2);

	cout << "First instance of " << p2 << " at "
		<< distance(p1,first_pos) << endl;
	cout << "Last instance of " << p2 << " at "
		<< distance(p1,last_pos) << endl;

	cout << "-----------------------" << endl;
}
```

Searching For Chars In A String

```
int main() {
	const char * str = "abcdeffedcba";
	demoStrchr(str,'a');
	demoStrchr(str,'b');
	demoStrchr(str,'c');
	demoStrchr(str,'d');
	demoStrchr(str,'e');
	demoStrchr(str,'f');
}
```

```
First instance of a at 0
Last instance of a at 11
-------------------------
First instance of b at 1
Last instance of b at 10
-------------------------
First instance of c at 2
Last instance of c at 9
-------------------------
First instance of d at 3
Last instance of d at 8
-------------------------
First instance of e at 4
Last instance of e at 7
-------------------------
First instance of f at 5
Last instance of f at 6
```


Searching For Substrings

```
int distance(const char * p1, const char * p2) {
	int count = 0;
	while(p1 != p2) {
		count++;
		p1++;
	}
	return count;
}
int main() {
	const char * data = "abcdefxyzghijklxyzmnopqrxyzuvwxyz";

	char * str = strstr(data,"xyz");
	cout << "Distance to first substring is: " << distance(data,str) << endl;
}
```

```
Distance to first substring is: 6
```


Tokenizing A String

```
int main() {
	char buffer[36];
	strcpy(buffer,"abc#def!ghi~jkl!mno#pqr!stu~vwz!yza");

	cout << "Tokens are:" << endl;
	char * str = strtok(buffer,"#!~");
	while(str != 0) {
		cout << '\t' << str << endl;
		str = strtok(0,"#!~");
	}
}
```

```
Tokens are:
	abc
	def
	ghi
	jkl
	mno
	pqr
	stu
	vwz
	yza
```

Pointers Part 3

Pointers and Structures

garth@ggilmour.com

Using Pointers With Structs

- A struct can be passed as a parameter via its address
  - Passing it by name alone copies the structure
    - This is OK if we don't want the client to see changes
    - But is very inefficient for large structures
- To access the structs fields we use the arrow operator
  - E.g. we say 'e_ptr->age' to access the age field of an Employee
  - We can also say '(*e_ptr).age' but this is awkward and ugly
- We can declare pointers to fields in the structure
  - E.g. 'int * age_ptr = &e.age' to directly point to the age field


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

struct Employee {
	int age;
	float salary;
	char name[10];
	char dept[10];
};
void useEmployee(struct Employee * e_ptr);
void printEmployee(struct Employee * e_ptr);

int main() {
	struct Employee e;
	useEmployee(&e);
	printEmployee(&e);
}
```

```c
void useEmployee(struct Employee * e_ptr) {
	e_ptr->age = 27;
	e_ptr->salary = 30000;
	strcpy(e_ptr->name, "Joe Smith");
	strcpy(e_ptr->dept,"Marketing");
}
void printEmployee(struct Employee * e_ptr) {
	printf("Employee %s in dept %s of age %d earning %.2f\n",
		e_ptr->name, e_ptr->dept, e_ptr->age, e_ptr->salary);
}
```

Project No 4

Converting Roman Numerals

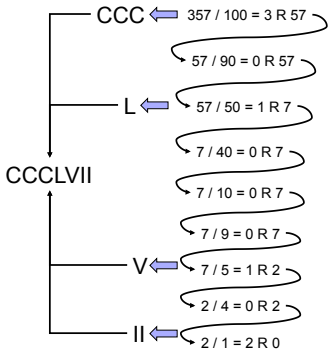
QUB – March 2013

garth@ggilmour.com

Roman Numerals

- Write a program to convert integers into Roman Numerals
  - You can assume there are no numerals above 'C'
- Guidelines are:
  - Use an array of nine instances of 'struct Numeral'
  - The structure should hold:
    - int decimalValue
    - char * numeral
  - Remember the modulus operator is '%'

Numeral	Value
C	100
XC	90
L	50
XL	40
X	10
IX	9
V	5
IV	4
I	1



Pointers Part 3

Functions and Heap Memory

garth@ggilmour.com

Using Pointers With Functions

- As with arrays the name of a function is its address
  - The location in read only memory of the compiled assembly
- The declaration of a function pointer is complex
  - For example 'int(*f_ptr)(int,double)' declares a pointer to a function that takes an int and a double and returns an int
- Brackets around 'f_ptr' are essential
  - To avoid declaring a function called 'f_ptr'
    - Which takes two parameters and returns a pointer
- To call a function through a pointer just use '( )'
  - E.g. given the declaration above 'f_ptr(27,3.6)'


```
#include<stdio.h>
void funcOne(int i) {
        printf("funcOne called with %d\n", i);
}
int funcTwo(int i, double d) {
        printf("funcOne called with %d and %f\n", i, d);
        return 7;
}
double funcThree(char * str) {
        printf("funcOne called with %s\n", str);
        return 7.89;
}
int main() {
        void (* ptr1)(int) = funcOne;
        int (* ptr2)(int, double) = funcTwo;
        double (* ptr3)(char *) = funcThree;

        ptr1(10);
        ptr2(11, 12.34);
        ptr3("ABCD");
}
```

```
funcOne called with 10
funcOne called with 11 and 12.340000
funcOne called with ABCD
```

Function Pointers and 'typedef'

- Function pointer declarations are hard to read
  - But can be simplified using the 'typdef' keyword
- Typedef declares an alternative name for a type
  - E.g. given 'typedef int * INT_PTR' the declaration 'INT_PTR ptr' makes 'ptr' a pointer to an integer
- Consider 'typedef void (*FUNC_INT_PTR)(int);'
  - Anything declared as being of type FUNC_INT_PTR will be a pointer to a function that takes an int and returns void
  - Note you could also say 'typedef void FUNC_INT(int);'
    - In which case the pointer declaration would be 'FUNC_INT * ptr;'


```
#include<stdio.h>

typedef void (*FUNC_INT_PTR)(int);

void funcOne(int i) {
	printf("funcOne called with %d\n", i);
}
void funcTwo(int i) {
	printf("funcTwo called with %d\n", i);
}
void funcThree(int i) {
	printf("funcThree called with %d\n", i);
}
void doCallback(FUNC_INT_PTR callback, int param) {
	callback(param);
}
int main() {
	FUNC_INT_PTR ptr1 = funcOne;
	FUNC_INT_PTR ptr2 = funcTwo;
	FUNC_INT_PTR ptr3 = funcThree;

	doCallback(ptr1,10);
	doCallback(ptr2,20);
	doCallback(ptr3,30);
}
```

```
funcOne called with 10
funcTwo called with 20
funcThree called with 30
```


Function Pointers in C++

- Function pointers are more complex in C++
  - Firstly because C++ supports overloading
    - Multiple functions can share the same name as long as they differ in the number, type or order of parameters
  - Secondly because C++ is an OO language
    - Objects contain fields and methods
      - Which may be declared public or private
    - Public methods are 'slots' on the outside of the object
      - A method pointer lets you call a slot on a particular object
- Consider the declaration 'void (*f_ptr)(int) = func;'
  - 'f_ptr' points to the 'func' that takes an int and returns void
  - As opposed to any other function with the same name

Function Pointers in C++

```
void func() {
	cout << "<void>func()" << endl;
}
int func(int i) {
	cout << "<int>func(<int>)" << endl;
	return i;
}
int func(int i,double d) {
	cout << "<int>func(<int><double>)" << endl;
	return i;
}
class MyClass {
public:
	int func() {
		cout << "<int>MyClass::func()" << endl;
		return 0;
	}
};
```

```
void main() {

	//The compiler can work out which function
	// we mean via the type of the pointer
	void(*f_ptr1)() = func;
	int(*f_ptr2)(int) = func;
	int(*f_ptr3)(int,double) = func;

	f_ptr1();
	f_ptr2(27);
	f_ptr3(27,8.3);

	//Show both ways of calling a method
	// via a pointer
	MyClass mc;
	MyClass *mc_ptr = &mc;
	int(MyClass::*m_ptr)() = &MyClass::func;
	(mc.*m_ptr)();
	(mc_ptr->*m_ptr)();
}
```


Constant Pointers

- Declaring parameters as constant pointers is useful
  - For parameters whose size is greater than that of a pointer
    - The value is no longer copied which improves performance
  - The 'const' signals to the client that the value wont be changed
- What 'const' means depends on where it is placed
  - 'int * const ptr' defines a pointer which is const
    - It always points to the same address but that data can be changed
  - 'const int * ptr' and 'int const * ptr' define pointers to const data
    - The address can be changed but not the data
  - 'const int * const ptr' defines a const pointer to const data

Allocating Heap Memory in C

- Pointers are mostly used to refer to heap memory
  - This is necessary when storage requirements can only be worked out a runtime, e.g. the number of records in a file
- Heap memory is allocated via 4 functions
  - Malloc, calloc, realloc and free
- Malloc returns a pointer to memory of a specified size
  - Where the size is determined by the 'sizeof' operator
  - E.g. 'struct P * ptr = (struct P *) malloc(sizeof(struct P));'
- Note that the returned memory is not initialized
  - You can set all bytes to the same value using 'memset'

Allocating Heap Memory in C

- Calloc returns an array of a specified type
  - E.g. 'int * ptr = (int *)calloc(10, sizeof(int));'
  - Unlike 'malloc' all the memory is initialized to zero
- Realloc resizes an array whilst preserving its contents
  - Either extra memory after the current block is allocated to you or the contents are copied into a larger block
  - E.g. 'int * new_ptr = (int *) realloc(old_ptr, new_size)'
- Free returns allocated memory to the heap
  - Calling free on a null pointer has no effect

Memory Problems in C

- Functions should not return pointers to local variables
  - When the frame is removed from the call stack the memory for the local variable is released and the pointer becomes invalid
- Its OK to return a pointer to heap memory
  - Although the client then has the burden of deleting it
- A wild pointer is a pointer holding a random address
  - Because you mistakenly think it has been initialized
  - Because you corrupted it via errors in pointer arithmetic
  - Because the memory was returned to the heap via 'free'
- Forgetting to delete memory is not technically an error
  - Instead it is a leak, which if repeated could exhaust the heap

Allocating Heap Memory in C++

- C++ allocates heap memory using the 'new' keyword
  - It provides the extra service of initializing the memory for you
- There are two versions of this operator
  - The 'new' operator allocates memory for a single value
    - For example 'int * i_ptr = new int;'
  - The 'new [ ]' operator allocates memory for an array of values
    - For example 'int * i_ptr = new int[10];'
- Heap memory must be managed carefully
  - To delete a single value use 'delete i_ptr;'
  - To delete an array of values use 'delete [ ] i_ptr;'

Memory Problems in C++

- If you use 'new [ ]' you must de-allocate with 'delete [ ]'
  - If you use 'new' then literally anything can happen
  - Typically only the first element will be removed
- Calling delete twice will corrupt the heap
  - However it is safe to call delete on a pointer set to zero
  - So always reset a pointer to zero after deletion if it is going to remain in scope as the program continues
- The 'new' operator will fail when the heap is used up
  - Prior to standardisation 'new' returned zero
  - In standard C++ an exception of type 'std::bad_alloc' is thrown
    - This is a class which inherits from 'std::exception'

Pass By Reference

Improved Parameter Passing

garth@ggilmour.com

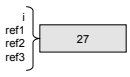
Introducing References

- The downside of pointers is the extra syntax
  - Continually having to remember to dereference the variable
- References were introduced as an alternative to pointers
  - Especially when it comes to avoiding pass by value
- A reference is an alternative name for a variable
  - It functions as an alias or synonym for the variable
- Unlike a pointer a reference has no separate existence
  - Once initialised it is indistinguishable from the variable is binds to
  - The address of the reference is the address of the variable

Declaring and Using References

```
//References don't work the same way as pointers...
int i = 27;
int & ref1 = i;
int & ref2 = ref1;
int & ref3 = ref2;

cout << "Address of i is " << &i << endl;
cout << "Addresses of references are " << &ref1 << " " << &ref2 << " " << &ref3 << endl;
cout << "Value of i is " << ref1 << " " << ref2 << " " << ref3 << endl;
```

```
Address of i is 0012FED4
Addresses of references are 0012FED4 0012FED4 0012FED4
Value of i is 27 27 27
```

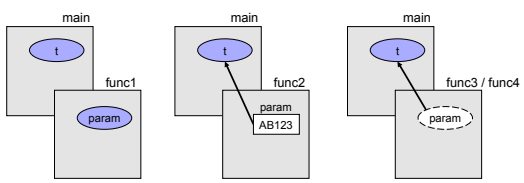


Benefits of References

- References provide the best of both worlds
  - The efficiency of passing by pointer combined with the convenience of pass by value
- The default way of passing non-basic parameters should be by constant reference
  - Use pass by value for basic types only
  - Use pass by pointer only to make it clear that the parameter lives in dynamically allocated memory
- The value returned from a function should be copied
  - Returning a reference or pointer usually leads to problems


```
//Pass by value - a copy is made
void func1(MyType param);
//Pass by pointer - the address is copied
void func2(MyType * param);
//Pass by reference - param is an alias
void func3(MyType & param);
//Pass by reference - param cannot be modified
void func4(const MyType & param);
```

```
int main() {
    MyType t;
    func1(t);
    func2(&t)
    func3(t);
    func4(t);
}
```



Using References

- Because a reference has no independent existence its declaration must also be its definition
  - As soon as it is declared it must be bound to a variable
- An initializer is not required when the reference:
  - Is a function parameter
  - Is the return type from a function
  - Is declared as a member of a class
  - Is declared with the 'extern' modifier
- A reference can be an alias for a pointer
  - Although mixing pointer and reference semantics is confusing

Combining References & Pointers

```
const char* str1 = "First test string";
const char* str2 = "Second test string";

//Swap the contents of two pointers using both
// pointer to pointers and references to pointers
void swapStrings(const char** str_ptr_ptr, const char* &str_ptr_ref) {

	const char* temp = *str_ptr_ptr;

	*str_ptr_ptr = str_ptr_ref;
	str_ptr_ref = temp;
}

void main() {
	swapStrings(&str1, str2);

	cout << "str1 now points to: " << str1 <<endl;
	cout << "str2 now points to: " << str2 <<endl;
}
```


Functions in C++

Resolving Overloaded Function Names

garth@ggilmour.com

Function Overloading

- In C code functions are linked by their name
  - You cannot have two functions with the same name
- In C++ multiple functions can share the same name
  - They must differ in the type, number or order of their parameters
  - We refer to this as the signature of the method
- Functions that do the same thing in different ways can be grouped together for our convenience
  - So we can have 'connect(string ip)' and 'connect(int ip)'
  - Rather than 'connectByString(string ip)' and 'connectByInt(int ip)'

Overload Resolution

- Matching function calls to their correct definitions is known as Overload Resolution
  - Developers need to be familiar with how it works
- Overload Resolution occurs in many places:
  - In function calls to overloaded functions
  - When the function call operator is used
  - When a function pointer is used
  - When an expression uses overloaded operators
  - Anytime a constructor is called to initialize an object
  - When copy and conversion constructors are called
  - The rules are also important in exception handling

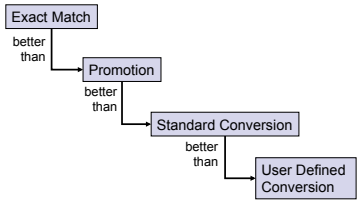
Resolving Overloaded Functions

- Name resolution in C++ is complex
  - It is often sidelined in introductory texts
- The procedure has three stages
  1. Identity the set of candidate functions
     - Those functions that match the name of the call
  2. Select the set of viable functions
     - Whose parameters match those used in the call
  3. Select the most viable function
     - The one whose parameters are the closest possible match
     - The complexity lies in finding the best possible match

Finding the Best Match

- Each parameter is ranked by asking:
  1. Is the parameter an exact match for the value supplied?
  2. Can the value be promoted to match the parameter?
  3. Can a standard conversion be used to convert the value?
  4. Is there a user defined conversion that can force a match?
- The result of this process determines the best match
  - Remember that the compiler views functions by signature
  - Each overloaded function just has the name part in common
  - Hence there is no problem with overloaded functions having the same return type - unlike virtual functions

Finding the Best Match



What is an Exact Match?

- Using a lvalue or rvalue of the specified type
  - E.g. calling 'func(int)' via 'func(intVar)' or 'func(27)'
- Converting an array name to a pointer
  - E.g. calling 'func(int *)' via 'func(intArray)'
- Converting a function name to a pointer
  - E.g. calling 'func(void (*fptr)(int))' via 'func(fooBar)'
- A qualification conversion
  - E.g. calling 'func(const float *)' using a non const float pointer
- Where the argument is a valid initializer
  - E.g. calling 'func(int &)' via 'func(myInt)'

Types of Conversion

- A standard conversion applies anywhere the range of the receiving type is greater
  - It is possible to convert from floating point to integer types
    - The fractional part of the number is lost
  - It is possible to convert from integer to floating point types
    - The converted value may lose some precision
  - Note that converting '0' to a pointer is a conversion
    - If you call 'func(0)' where the candidates are 'func(float *)' and 'func(int)' the latter is always selected as an exact match
- User defined conversions are often constructors
  - The compiler will match 'func(27)' to 'func(MyClass)' if class 'MyClass' has a conversion constructor that takes an integer
  - This is something you want to be very careful about allowing

Namespaces

Partitioning C++ Code

garth@ggilmour.com

Namespaces

- Namespaces are fundamental to standard C++
  - They split your code into multiple declarative regions
  - A namespace may span any number of Translation Units
  - This is essential for any modern language
    - Where programs with hundreds of classes are common
    - UML packages map directly to C++ namespaces
- Unnamed namespaces replace static functions
  - An unnamed namespace is given an unknown name that is guaranteed to be different from all others in the program
  - Declarations inside the namespace cannot be discovered outside the current TU and hence all have internal linkage

Namespace Definitions

- Namespace members must be declared inside its scope
  - Definitions can be placed elsewhere
    - By qualifying the member with appropriate name
    - For example 'void MyNamespace::func(int i)'
  - The declaration must already be visible and the definition must occur inside an enclosing namespace
  - Unqualified names are said to be in the 'global namespace'
- A lesser known C++ feature is the namespace alias
  - For example 'namespace WRUS = Widgets_Are_Us'
- The standard libraries use the 'std' namespace
  - Header files drop the '.h' postfix to coexist with earlier versions
  - So prefer '#include <string>' to '#include<string.h>'

Using Declarations

- A using declaration introduces extra symbols into the current scope
  - Typically these symbols are from other namespaces or base classes
- The declaration typically occurs at the top of a '.cpp' or '.h' file
  - However it can also be used in class declarations

Declaration	Meaning
using namespace std	Bring in all visible symbols from namespace std
using ::foobar	Bring in symbol 'foobar' from the global namespace
using A::func	Bring in 'func' from the namespace or base class A
using A::B::C::func	Bring in the symbol from a nested namespace

Using Declarations

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

//bring in all symbols from std namespace
using namespace std;

namespace A {
   void print(string str) { cout << str << endl; }
}
namespace B {
   //bring in a single symbol from namespace A
   using A::print;

   class Base {
   public:
      void funcOne(int) {  print("Base::funcOne"); }
      void funcTwo(int) { print("Base::funcTwo"); }
   };
```

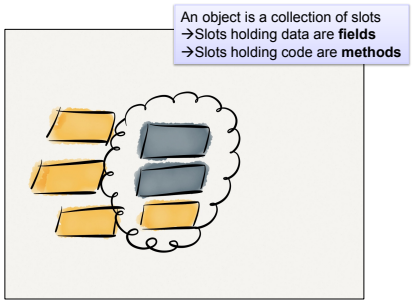
```
   class Derived : Base {
      //bring in funcOne from base class
      using Base::funcOne;
   public:
      void funcOne(char) {
         print("Derived::funcOne");
         //calls base class function
         funcOne(7);
      }
      void funcTwo(char) {
         print("Derived::funcTwo");
         //causes infinite recursion
         funcTwo(7);
      }
   };
}
```

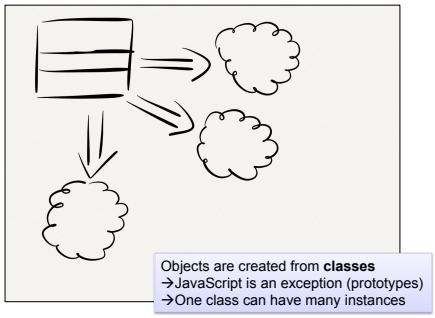
A Cross-Language Introduction to OO

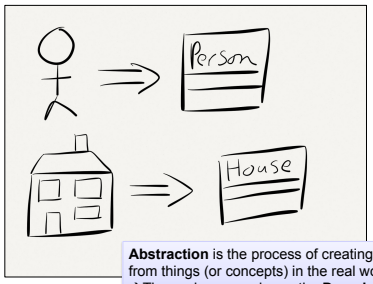
The Core Concepts

QUB – March 2013

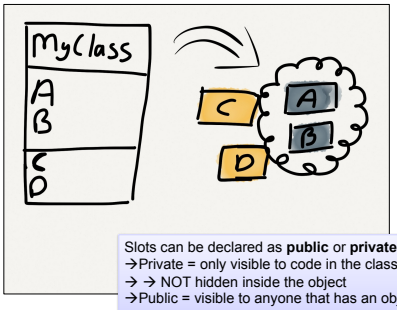
garth@ggilmour.com





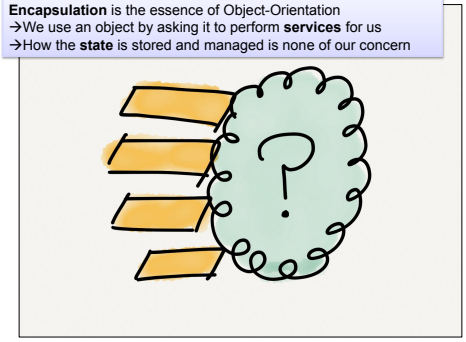


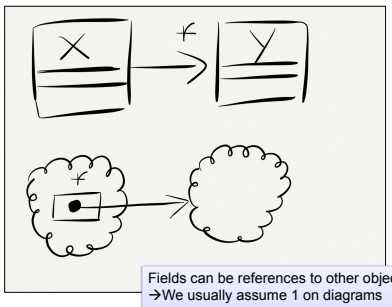
Abstraction is the process of creating classes from things (or concepts) in the real world
→These classes make up the **Domain Model**



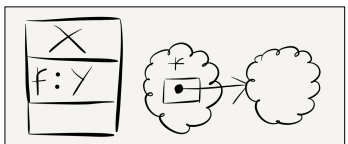
Slots can be declared as **public** or **private**
→Private = only visible to code in the class
→ → NOT hidden inside the object
→Public = visible to anyone that has an object

Encapsulation is the essence of Object-Orientation
→We use an object by asking it to perform **services** for us
→How the **state** is stored and managed is none of our concern

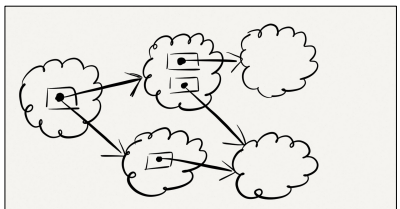




Fields can be references to other objects
→We usually assume 1 on diagrams
→More than 1 would require a **collection**

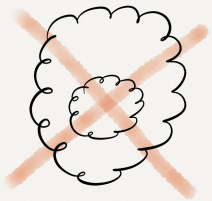


This is identical to the previous diagram
→ Except that because we are not drawing a separate box for class 'Y' we are unable to show its details (do this for built in classes)

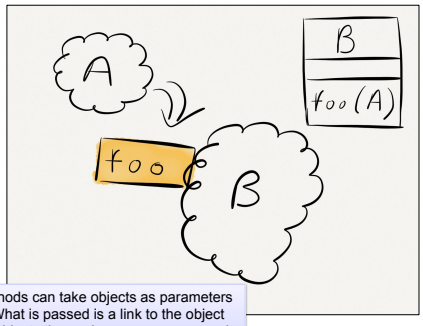


A running program is made up of a graph of objects
→ We get good at visualizing this through practice!!!
→ Unreachable objects get garbage collected

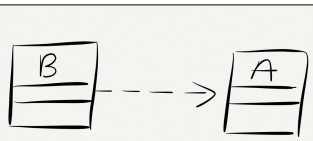
Objects never contain other objects



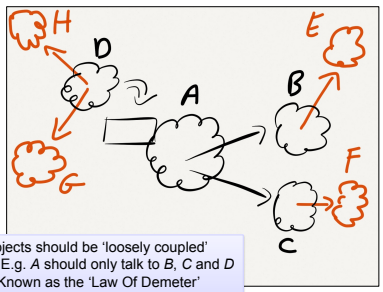
Except in C++ where it is a huge pain...



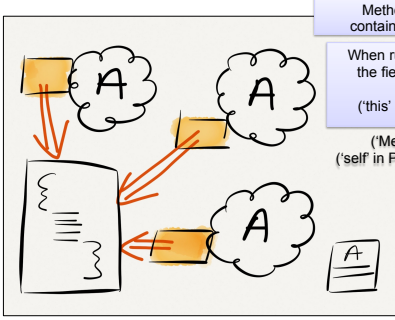
Methods can take objects as parameters
→ What is passed is a link to the object
→ Objects themselves are never moved



If one or more methods of 'B' take an 'A' as a parameter then the relationship is called a **dependency**

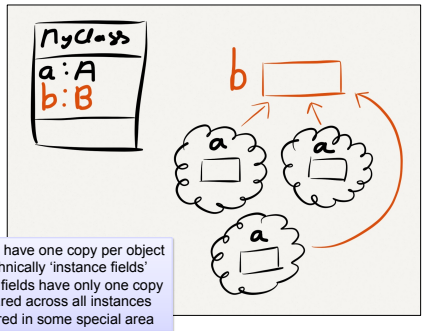


Objects should be 'loosely coupled'
→ E.g. *A* should only talk to *B*, *C* and *D*
→Known as the 'Law Of Demeter'
→→ 'Only talk to your friends'



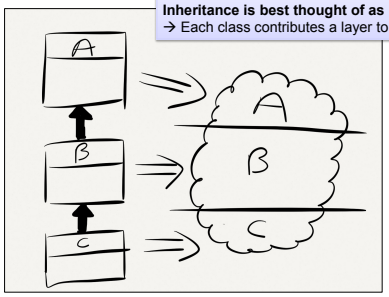
Methods are slots that contain the address of a

When run the code uses the fields of the current object
('this' in C++, Java and C#)
('Me' in Visual Basic)
('self' in Python and Ruby)

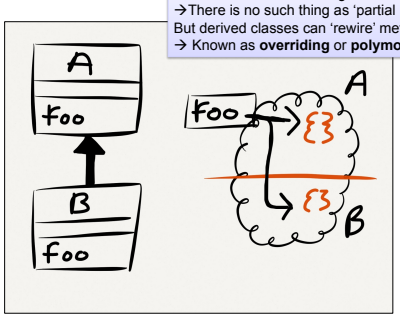


Fields have one copy per object
→Technically 'instance fields'
Static fields have only one copy
→Shared across all instances
→Stored in some special area

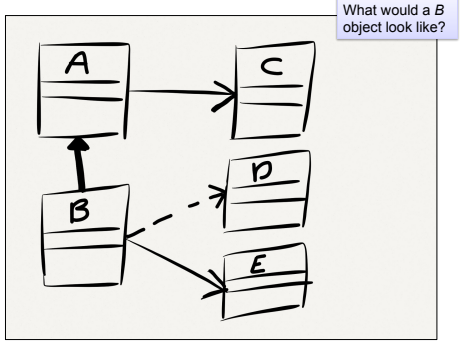
Classes can inherit from one another
→This prevents duplication of declarations
Inheritance is best thought of as layering
→ Each class contributes a layer to the object

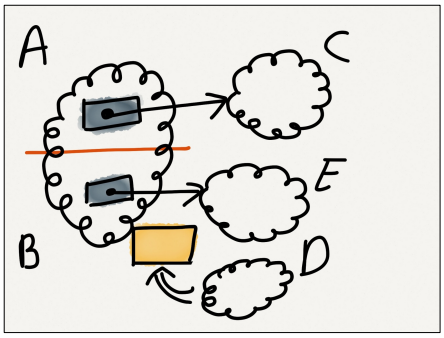


Inheritance is all or nothing
→There is no such thing as 'partial inheritance'
But derived classes can 'rewire' methods
→ Known as **overriding** or **polymorphism**



What would a *B* object look like?





Class Design Part 1

Declaring & Defining Classes

garth@ggilmour.com

Procedural Programming

- C style procedural programming uses:
  - Data structures to model problem domain entities
  - Functions which manipulate the data structures
- This has one key advantage
  - It lets you write lean systems with very efficient code
- There are many disadvantages
  - Large applications become very complex
  - Many functions operate on the same data structures
  - Developers lose track of all the functions, their intended purpose and how they are supposed to be used
    - Especially over several generations as the system is maintained

Procedural Programming

```
struct Employee {
      int Age;
      float Salary;
      char Name[10];
      char Dept[10];
};
```

Age		Salary	
N	a	m	e
D	e	p	t

```
void useEmployee(Employee & e) {
      e.Age = 27;
      e.Salary = 30000;
      strcpy(e.Name, "Joe Smith");
      strcpy(e.Dept,"Marketing");
}
```

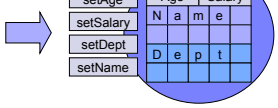
27		30000	
J	o	e	S
m	i	t	h	\0
M	a	r	k	e
t	i	n	g	\0

Object Oriented Programming

- Object oriented programming is about removing the separate organization of data and functions
  - An object is something which contains data and has functions which manipulate that data
- A class is a specification for building objects
  - We can have many instances of the class in memory
  - Variables have a type but an object has a class
- OO systems are made up of collaborating objects
  - Objects talk to each other by calling each others functions
  - We refer to this as 'message passing'

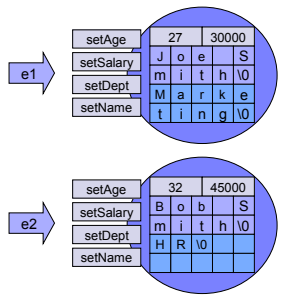
Object Oriented Programming

```
class Employee {
      int Age;
      float Salary;
      char Name[10];
      char Dept[10];

      void setAge(int pAge) {
            Age = pAge;
      }
      void setSalary(float pSalary) {
            Salary = pSalary;
      }
      void setName(char* pName) {
            strcpy(Name, pName);
      }
      void setDept(char* pDept) {
            strcpy(Dept,pDept);
      }
};
```



Object Oriented Programming

```
Employee e1;
Employee e2;

e1.setAge(27);
e1.setSalary(30000);
e1.setName("Joe Smith");
e1.setDept("Marketing");

e2.setAge(32);
e2.setSalary(45000);
e2.setName("Bob Smith");
e2.setDept("HR");
```



Terminology

- Classes contain members
  - Members containing data are fields
  - Members containing code are methods
- An instance of a class is an object
  - A local object is allocated on the stack
- Functions which are not methods are free
  - Free functions exist outside of class declarations
- In some cases the type of an entity is uncertain
  - The apparent type is what the entity appears to be
  - The inherent (or actual) type is what the entity is

Class Declarations

- Inside a class declaration are the member declarations
  - These can be zero or more fields, methods and constructors
  - A single destructor may optionally be specified
  - Any class member must be declared inside the class
- Definitions can go inside or outside the declaration
  - They go outside if the definition is qualified with the class name
  - The declaration is usually placed separately from the definitions
    - Into header and source files e.g. 'Employee.h' and 'Empoyee.cpp'
    - Template class declarations are an exception to this rule
  - Although not required one class per header/source file is useful
    - It makes classes very easy to find

Class Accessibility

- Everything declared in a class has an accessibility
  - How visible it is to other parts of the program
- C++ has three accessibilities
  - Public members are accessible to everyone
  - Private members are hidden from everyone
  - Protected members are available to derived classes
- The default accessibility is private
  - This is what we want for fields and helper methods
- Accessibility is indicated via an access specifier
  - The label 'public:' 'private:' or 'protected:'
  - An accessibility remains in force until changed

Encapsulation

- Encapsulation is the most critical OO concept
  - Without it everything else falls apart
- Clients should have no access, either direct or indirect, to the data held inside your object
  - Any public fields must be constant values
  - No method should return a pointer or reference to a field
- Methods should be designed from a clients perspective
  - The name should indicate what the method does
  - A method should either query or change the objects state
    - It is good design not to have methods which do both
- Encapsulation does have practical limitations
  - A class cannot display itself, print itself, email itself etc…

Constructors

- A constructor is a used to initialize an object
  - A function with the name of the class and no return type
- Your class can have many constructors
  - One for each valid way of creating an object
  - Unfortunately C++ constructors cannot call one another
- A default constructor is written for you
  - If and **only** if you don't declare any constructors of your own
- When you create an object the parameters you pass must match against a viable constructor call
  - To call the default constructor you say 'MyClass mc;'
  - Not the expression 'MyClass mc();' which declares a function

Class Employee

```
/* Class declaration - normally goes in header file */
class Employee {
//Public members can be used by anyone
public:
	//Constructors
	Employee();
	Employee(const string& name);
	Employee(const string& name, const string& address);
	Employee(const string& name, const string& address, double basicSalary);
	//Destructor
	~Employee();
	//Methods
	void printDetails();

//Private members are hidden from everyone
private:
	//Fields
	string name;
	string address;
	double basicSalary;
};
```


Class Employee

```
/* Class definition - normally goes in cpp file */
Employee::Employee() {
	cout << "default constructor" << endl;
	name = "NO_NAME";
	address = "NO_ADDRESS";
	basicSalary = 0;
}
Employee::Employee(const string& name) {
	cout << "single parameter constructor" << endl;
	this->name = name;
	address = "NO_ADDRESS";
	basicSalary = 0;
}
Employee::Employee(const string& name, const string& address) {
	cout << "two parameter constructor" << endl;
	this->name = name;
	this->address = address;
	basicSalary = 0;

}
```


Class Employee

```
Employee::Employee(const string& name, const string& address, double basicSalary) {
	cout << "three parameter constructor" << endl;
	this->name = name;
	this->address = address;
	this->basicSalary = basicSalary;
}
Employee::~Employee() {
	cout << "Destructor of Employee " << name <<endl;
}
void Employee::printDetails() {
	cout << "Employee: " << name << " from " << address << " earning " << basicSalary;
}
int main() {
	Employee e;
	e.printDetails();
	Employee e2("Joe Bloggs");
	e2.printDetails();
	Employee e3("Fred Flintstone","10 Arcatia Drive");
	e3.printDetails();
	Employee e4("Barney Rubble","10 Prospect Park",20000);
	e4.printDetails();
}
```

The ‘this’ Pointer

- Every object has a built in pointer to itself
  - Represented by the keyword ‘this’ (for *this object*)
- The ‘this’ pointer has the type of the current class and refers to the data structure of fields inside the object
  - It can only be used in methods and constructors
  - If the method is ‘const’ or ‘volatile’ then so is the pointer
- The main use of ‘this’ is to resolve scoping issues
  - If the field called ‘x’ is hidden by a local variable or parameter called ‘x’ then the expression ‘this->x’ can be used to reveal it
  - Otherwise the expressions ‘x’ and ‘this->x’ are identical
  - Some teams prefix or postfix all field names with an underscore

The Destructor

- Destructors are the reverse of constructors
  - They are called at the end of an objects lifetime
- A class can only have a single destructor
  - It is declared as ‘~*ClassName*’ with no parameters
  - An empty destructor is generated by default
- The destructor is automatically called when:
  - A local object goes out of scope (e.g. end of method)
  - You call delete on a pointer to an object
- The usual practise with resources is:
  - Acquire the resource in the constructor
  - Release the resource in the destructor

Structures and Nested Classes

- Classes and structs are almost identical
  - A struct can have the same features as a class
  - The only difference is that the default accessibility is public
- Using ‘struct’ indicates that a class is just a data holder
  - As opposed to a proper object with state and behaviour
- Class declarations can be nested inside one another
  - Names of nested classes are local to the enclosing class
  - Nested classes can use static methods, enumerations and types defined in the enclosing class but have no other rights
- Nesting is a good way to represent a helper class
  - However nested classes cannot be used inside templates

Project No 5

Creating Classes

garth@ggilmour.com

Class Declarations

- Create a 'Student' class with these fields
  - string name;
  - string courseTitle;
  - int yearOfStudy;
  - double marks[10];
- The class should have:
  - Appropriate constructors
    - A default and one or more that takes values
  - A 'print' method
  - An 'average' method
    - That calculates the mean of all the marks

Class Design Part 2

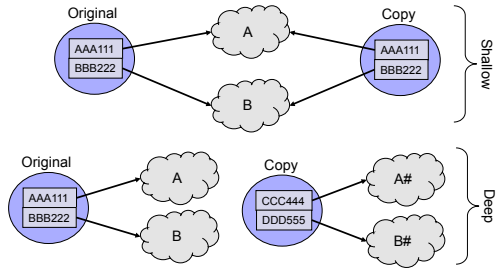
Understanding Local Objects

garth@ggilmour.com

Copy Constructors

- A copy constructor creates a new instance of your class
  - Out of an existing instance of the same class
  - It is declared as 'MyClass::MyClass(const MyClass& obj)'
- The compiler writes a copy constructor for you
  - This performs a simple bitwise copy of the fields
- The compiler will call the constructor automatically
  - Every time you pass or return an object by value
- Classes with pointers usually need bespoke constructors
  - The compilers constructor only generates a shallow copy
  - Normally you want to make a deep copy by making a copy of what the pointer(s) point to, rather than just the pointer itself

Shallow Verses Deep Copies



The Syntax of a Copy Constructor

```
class MyClass {
public:
    MyClass(int p1,double p2) {
        field1 = p1;
        field2 = p2;
    }
    MyClass(const MyClass & other){
        cout << "MyClass copy constructor" << endl;
        field1 = other.field1;
        field2 = other.field2;
    }
    void print() {
        cout << "Values are: " << field1 << " " << field2 << endl;
    }
private:
    int field1;
    double field2;
};
```

The Syntax of a Copy Constructor

```
void doPrint(MyClass param) {
    param.print();
}

int main() {
    MyClass mc1(7,3.4);
    mc1.print();

    MyClass mc2 = mc1;
    MyClass mc3(mc1);

    mc2.print();
    mc3.print();

    doPrint(mc1);
    doPrint(mc2);
    doPrint(mc3);
}
```

```
Values are: 7 3.4
MyClass copy constructor
MyClass copy constructor
Values are: 7 3.4
Values are: 7 3.4
MyClass copy constructor
Values are: 7 3.4
MyClass copy constructor
Values are: 7 3.4
MyClass copy constructor
Values are: 7 3.4
```


Conversion Constructors

- A conversion constructor creates an instance of your class out of an instance of another type
  - For example creating an Employee out of a Person object
- These constructors will be called by the compiler
  - If the only viable way to call a function is to convert the value passed by the caller into an instance of your class
  - Be wary of conversion constructors with a single parameter
- Watch out for default parameters
  - If your constructor has multiple parameters with default values it can still be called implicitly by the compiler

Conversion Constructors

```
class MyClass {
public:
    MyClass(int p1,int p2) {
        field1 = p1;
        field2 = p2;
    }
    MyClass(int p1) {
        field1 = p1;
        field2 = p1 + 1;
    }
    void print() {
        cout << "Values are: " << field1
             << " and " << field2 << endl;
    }
private:
    int field1;
    int field2;
};
```

```
void doPrint(MyClass param) {
    param.print();
}

int main() {
    MyClass mc1(7,6);
    doPrint(mc1);

    doPrint(167);
}
```

```
Values are: 7 and 6
Values are: 167 and 168
```

Operator Overloading

- C++ lets you change how the built in operators of the language behave for your class
  - So you can write 'obj1 + obj2' or 'cout << obj3'
- This is useful for classes that model maths concepts
  - Points, vectors, complex numbers etc…
  - For other classes its use should be limited
- Some templates require that your class has operators
  - For example 'operator==' and 'operator<' are used to compare and sort values inside a collection like a list
- Almost all operators in C++ can be overloaded
  - Including esoteric ones like the logical and cast operators

The Assignment Operator

- An assignment operator is written for you
  - This is called the copy assignment operator
  - It is declared as 'X& X::operator=(const X&)'
    - You use this when you write 'obj1 = obj2'
- Assignment operators resemble copy constructors
  - The default operation does a bitwise copy
  - You need to create your own for a deep copy
- Your class can have many assignment operators
  - For example 'X& X::operator=(int i)' so you can say 'obj1 = 12'


```
class MyClass {
public:
    MyClass(int p1,double p2) {
        field1 = p1;
        field2 = p2;
    }
    MyClass(const MyClass & other) {
        cout << "MyClass copy constructor" << endl;
        field1 = other.field1;
        field2 = other.field2;
    }
    MyClass & operator=(const MyClass & rhs) {
        cout << "MyClass assignment operator" << endl;
        field1 = rhs.field1;
        field2 = rhs.field2;
        return *this;
    }
    void print() {
        cout << "Values are: " << field1 << " " << field2 << endl;
    }
private:
    int field1;
    double field2;
};
```

```
//COPY CONSTRUCTOR CALLED
void doPrint(MyClass param) {
    param.print();
}

int main() {
    MyClass mc1(7,3.4);
    mc1.print();

    //In C++ you can say 'int i = 7;' or 'int i(7);'
    MyClass mc2 = mc1;    //COPY CONSTRUCTOR
    MyClass mc3(mc1);     //COPY CONSTRUCTOR

    MyClass mc4(45,67.8);

    //Equivalent to mc2.operator=(mc4)
    mc2 = mc4;          //ASSIGNMENT OPERATOR
    mc2.print();

    //Equivalent to mc4.operator=(mc3.operator=(mc1));
    mc4 = mc3 = mc1;
    mc4.print();
}
```


The Assignment Operator

- The operator returns a reference to the current object
  - So clients can write 'obj1 = obj2 = obj3'
  - The easiest way to do this is by returning '*this'
- You should always assign values to all fields
  - So that the object on the left hand side does not retain any fragments of its original value

```
//the two lines below are equivalent
obj1 = obj2 = obj3 = obj4;
obj1.operator=(obj2.operator=(obj3.operator=(obj4)));
```


Class Design Part 3

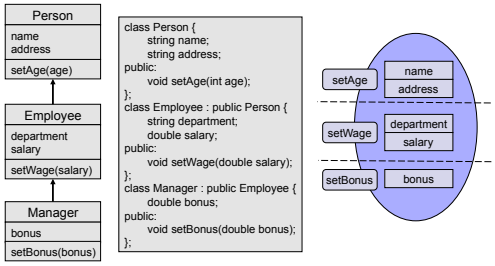
Inheritance and Polymorphism

garth@ggilmour.com

Inheritance

- Declaring related types results in duplicated code
  - E.g. if we declare 'Person', 'Employee' and 'Manager' separately there will be many duplicated fields and functions
- Inheritance lets a type be defined out of existing types
  - If you inherit from a type you obtain a copy of all its members
- The golden rule of inheritance is 'IS-A'
  - You only inherit if the derived type IS-A-KIND-OF the base type
- With inheritance it helps to view your objects as layered
  - There is one layer for each level in the inheritance hierarchy
  - The C++ specification refers to 'subobjects' rather than layers

Inheritance



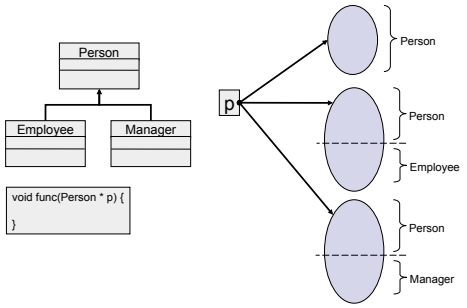
Inheritance and Accessibility

- Understand how private fields work with inheritance
  - Many developers confuse this relationship
- Private fields are part of the derived object
  - Being private does not prevent them from being inherited
- Private fields cannot be accessed from derived methods
  - The compiler ensures that only base class methods are allowed to access private fields of the base class
  - Derived methods can call base methods to use private fields
- Fields can be made protected to be seen by subclasses
  - Protected fields have a mixed reputation amongst developers
  - Some developers recommend only using protected helper functions to let derived classes manipulate base class fields

Inheritance and Pointers

- A pointer or reference of type ‘Base’ can refer to an instance of any class derived from ‘Base’
  - Base class pointers can address derived objects
- The compiler will only let you access the members in the layer corresponding to the pointer type
  - The other members are still there but cannot be accessed without a cast for safety reasons
- Derived objects should not be passed by value where a base object is expected (functions and arrays)
  - Only the base part of the object is copied and the rest is lost
  - This is referred to as ‘slicing’ the object

Inheritance and Pointers



Constructors of Derived Classes

- Constructors are not inherited
  - A class initialization cannot be mapped to a base constructor
  - It is each classes responsibility to define the constructors it needs to instantiate objects
- Derived constructors must call base constructors
  - It is important that an object be initialised from the top down
  - Because derived fields may use base fields in their definitions
  - The first thing a derived constructor should do is call a base one
- Base constructors are called in a initializer list
  - The default constructor will be called if you do not use this

Constructors of Derived Classes

```
class Base {
public:
   Base();
   Base(int i, float f);
private:
   int ival;
   float fval;
};

class Derived : public Base {
public:
   Derived();
   Derived(int i, float f);
   Derived(int i, float f, double d);
private:
   double dval;
};
```

```
Base::Base() {
   ival = 0; fval = 0;
}
Base::Base(int i, float f) {
   ival = i;
   favl = f;
}
//default constructor called automatically
Derived::Derived() {
   dval = 0;
}
Derived::Derived(int i, float f) : Base(i,f) {
   dval = 0;
}
Derived::Derived(int i, float f, double d) : Base(i,f) {
   dval = 0;
}
```


Polymorphism

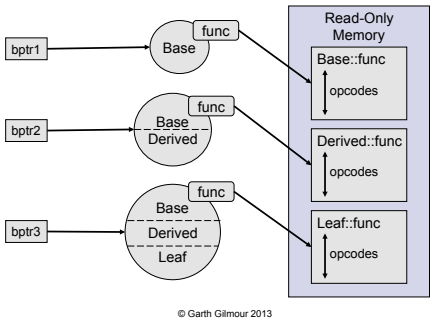
- Polymorphism is the final pillar of object orientation
  - After abstraction, encapsulation and inheritance
- Derived classes can override base class methods
  - By declaring a function which has exactly the same signature as the base class method
  - The base class method must be declared as 'virtual'
- The base slot then points to the derived implementation
  - The method to be called depends on the type of the object
  - Not on the type of pointer or reference that the client has

Polymorphism

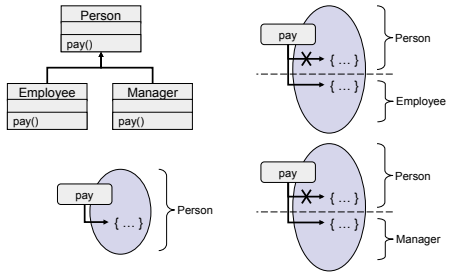
```
class Base {
public:
      virtual char* func(){
            return "base version of func";
      }
};
class Derived : public Base {
public:
      char* func(){
            return "derived version of func";
      }
};
class Leaf : public Derived {
public:
      char* func(){
            return "leaf version of func";
      }
};
```

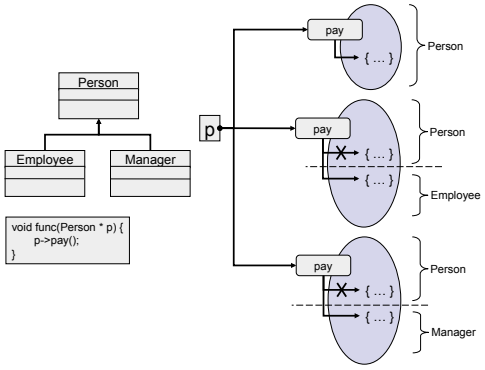
```
int main() {
      Base b;
      Derived d;
      Leaf e;

      Base* bptr1 = &b;
      Base* bptr2 = &d;
      Base* bptr3 = &e;

      //calls Base::func
      cout << bptr1->func() << endl;
       //calls Derived::func
      cout << bptr2->func() << endl;
       //calls Leaf::func
      cout << bptr3->func() << endl;
}
```

Polymorphism



Polymorphism





Polymorphism

- Polymorphism simplifies the job of clients
  - The client calls the base class method and lets the object use the implementation most appropriate to it
- This is especially useful in frameworks
  - E.g. extend the 'Button' class and override 'onPush'

```
//Utility method to promote all employees.
//The list may contain many different subclasses of Employee but
//  thanks to polymorphism we can treat them all the same
void promoteAllEmployees(List& employeeList) {
      while(employeeList.hasNext()) {
            Employee * e_ptr = employeeList.getNext();
            e_ptr->promote();
      }
}
```


Polymorphism in GUI Libraries

```
class Widget {
public:
      virtual void display();
private:
      int x;
      int y;
      int length;
      int height;
};
```

```
class Button : public Widget {
public:
      void display();
};
```

```
class Window : public Widget {
public:
      void display() {
         drawBorders();
         drawTitleBar();
         for(int i=0;i<numComponents;i++) {
           components[i]->display();
         }
      }
private:
      Widget ** components;
      int numComponents;
};
```


Static Members

- Some data is associated with classes rather than objects
  - The data concerns the abstraction the class represents
  - There should be one copy regardless of the number of objects
- This data is declared as being 'static'
  - A different meaning from the C usage for internal linkage
- Static fields need to be defined outside the class
  - They will be initialized before the main function is called
    - For example 'int MyClass::staticField = 0'
  - Note that the order in which static fields are initialized in different translation units is implementation dependant
- A method which only uses static data is a static method
  - It can be called using the class name rather than via an object

Static Members

```
class Test {
private:
        static int createdCounter;
        static int existingCounter;
public:
        static int numCreated();
        static int numObjects();
        Test();
        Test(const Test& rhs);
        ~Test();
};
int Test::createdCounter = 0;
int Test::existingCounter = 0;

int Test::numObjects() {
        return existingCounter;
}
int Test::numCreated() {
        return createdCounter;
}
```

```
Test::~Test() {
        existingCounter--;
}
Test::Test() {
        createdCounter++;
        existingCounter++;
}
Test::Test(const Test& rhs) {
        createdCounter++;
        existingCounter++;
}
int main() {
        vector<Test> v;
        for(int i=0; i< 100; i++) {
                Test t;
                v.push_back(t);
        }
        cout << "Created: " << Test::numCreated() << endl;
        cout << "Current: " << Test::numObjects() << endl;
}
```


Project No 6

Creating Class Hierarchies

garth@ggilmour.com

Inheritance Declarations

- Create a 'GraduateStudent' class
  - Which inherits from 'Student'
- The class should contain these fields:
  - int teachingHours;
  - string supervisor;
- The class should also have:
  - An appropriate constructor
  - A 'print' method
  - Anything else you want…
- Test your class
  - Through derived and base pointers

Class Design Part 4

Advanced Concepts

garth@ggilmour.com

Casting in C++

- Most C++ coders use the C syntax for casting
  - It can perform any kind of cast, despite being easy to ignore
- C++ provides four casting operators
  - Each performs a specific function and is visually distinctive
  - They should be preferred to the traditional syntax

Cast Operator	Description
const_cast<*TYPE*>	Used to remove 'constness' from a variable
dynamic_cast<TYPE>	Used to perform a downcast (e.g. Employee * to Manager *)
reinterpret_cast<TYPE>	Used to convert between unrelated types (e.g. storing memory addresses in integers)
static_cast<*TYPE*>	Used for all other types of conversion

Issues With Objects

- Many OO problems in C++ result from pass by value
  - Which is why all Java and C# objects are heap based
- Passing or returning a value creates a copy
  - The compiler automatically calls the copy constructor
- Storing a derived object in a base array creates a slice
  - Only the base part of the object is copied
- You can require that an object be created on the heap
  - By making all the constructors and/or the destructor private
    - Just making the destructor private is the simplest option
  - This ensures the compiler cannot put an instance on the stack
  - However it also makes it very difficult to design an derived class

Slicing Objects

```
class Base {
public:
	virtual void func();
};
class Derived : public Base {
public:
	void func();
};
void Base::func() {
	cout << "Base version of func called..."
		<< endl;
}
void Derived::func() {
	cout << "Derived version of func called..."
		<< endl;
}
```

```
void test(Base b) {
	b.func();
}

int main() {
	Derived d;

	//show slicing in arrays
	Base tstarray[10];
	tstarray[0] = d;
	tstarray[0].func();

	//show slicing in parameters
	test(d);
}
```


The Explicit Keyword

- The keyword explicit can only be used with constructors
  - It signifies that a constructor is an 'explicit constructor'
- Explicit constructors can only be used in the initialization of a new object or as the result of an explicit cast
  - The compiler cant use them for pass by value
  - This makes your conversion constructors safer

```
class Tester {
	int value;
public:
	explicit Tester(int i) {
		value = i;
	}
};
void func(Tester t) {}
```

```
int main() {
	Tester t1(7);		//OK - direct initialization
	Tester t2 = (Tester)8;	//OK - explicit cast
	Tester t3 = 9;		//INVALID - implicit cast
	func(10);		//INVALID - implicit cast
}
```


What Accessibility Means

- Many developers misunderstand accessibility
  - They view it in terms of objects rather than classes
- Accessibility is granted to classes not objects
  - Private fields can only be used by members of the current class
    - But these methods can be static or can be found in any instance
  - Private does NOT mean accessible only from the current object
    - Otherwise how could you write copy constructors or operators?
- Accessibility is a compile time concept
  - The compiler checks that members are being accessed correctly
  - These checks are not repeated at runtime

Friendly Functions

- Classes can declare classes and methods to be friends
  - For example 'friend class MC;' or 'friend void func();'
- Friends can to access all the members of the class
  - Friendship is not inherited by derived classes
- Friendliness is a one way concept
  - Your class does not have special access to its friends members
- Friend functions are sometimes unavoidable
  - For example if you want to be able to write 'cout << myObject' then the global 'operator<<' must be a friend of your class
  - Your should not use friend as part of your design
    - It is an obvious violation of encapsulation


```
class X;
int test(X& x);
class Z {
public:
      int test(X& x);
};
class Y{
public:
      int test(X& x);
};
class X {
private:
      int i;
      int func(){return 27;}
      //friend declarations
      friend int test(X& x);
      friend int Y::test(X& x);
      friend class Z;
      //this rarely used syntax defines a global
      // function called test2 which is a friend of X
      friend int test2(X& x) {
            return x.func();
      }
public:
      X():i(17){}
};
```

```
int test(X& x) {return x.i;}
int Y::test(X& x){return x.i;}
int Z::test(X& x){return x.func();}

int main(){
      X x;
      Y y;
      Z z;

      cout << "Values returned are: "
            << test(x)   << " "
            << test2(x) << " "
            << y.test(x) << " "
            << z.test(x) << endl;
}
```


Initializer Lists

- A class may have a member initialization list
  - Which specifies initial values for one or more fields
- The list is run as the object is being created
  - The constructor body executes after the object has been created
  - This makes using the list slightly more efficient
- Initialization lists are sometimes mandatory
  - Where the class has fields which are const or references
  - This is the only place in which they can be initialized
- Lists have one counter intuitive feature
  - The order of the initializations in the list is unimportant
  - The initializations will occur in the order in which the fields were declared inside the class declaration

Initialization Order

```
class person {
private:
      const int ageNextYear;
      const int ageNow;
public:
      person(int age);
      void print();
};
// despite the order of the initializations in the list ageNextYear will be
//   initialized before ageNow because its declaration appears first
person::person(int age) : ageNow(age), ageNextYear(ageNow + 1) {}

void person::print() {
      cout << "Person aged " << ageNow << " now and "
            << ageNextYear << " next year " << endl;
}
int main() {
      person dave(29);
      // prints "Person aged 29 now and -858993459 next year"
      dave.print();
}
```


Constant Methods

- A method can be declared with the 'const' modifier
  - Which means that the 'this' pointer will be const as well
  - Hence it is not possible to modify any of the fields
- Using constant methods is good OO design
  - It clearly distinguishes query from modifier methods
- Methods can be overloaded based on const
  - If the object has been declared as const then the const overload will be called, otherwise the non-const version
- Const methods don't guarantee immutability
  - A const method can still return pointers to internal data
  - The developer can still cast away the constness

Constant Methods

```
class MyClass {
public:
    MyClass(int p) : i(p) {}
    int get() const {
        cout << "const version of get" << endl;
        return i;
    }
    int get() {
        cout << "non const version of get" << endl;
        return i;
    }
private:
    int i;
};
```

```
void printOne(const MyClass & param) {
    cout << param.get() << endl;
}
void printTwo(MyClass & param) {
    cout << param.get() << endl;
}
void printThree(const MyClass * param) {
    cout << param->get() << endl;
}
void printFour(MyClass * param) {
    cout << param->get() << endl;
}
int main() {
    MyClass mc(30);
    printOne(mc);
    printTwo(mc);
    printThree(&mc);
    printFour(&mc);
}
```

```
const version of get
30
non const version of get
30
const version of get
30
non const version of get
30
```

The Mutable Keyword

- Mutable makes it easier to write const methods
  - A mutable field is exempt from the const specifier applied to a method or object
  - You can modify a mutable field without sacrificing the 'const-ness' of the rest of the object
- With mutable you define what const is for your class
  - Fields used as 'scratch memory' or that contain values that are not part of an objects identity can be changed
- A field cannot be declared as both mutable and const
  - NB 'mutable const int * ptr' is fine because it is the pointer which is mutable, the expression 'mutable int * const ptr' is invalid

Forward References

- Headers files common include many other headers
  - This complicates maintenance and slows down the build
- Many header file includes can be removed
  - The compiler does not need the size of a type if the class declaration only contains pointers or references to that type
  - The compiler only needs to see a declaration of the type
  - Most parameters should already be const references
- This often removes headers for standard libraries
  - Many of which have been added via cut and paste

Forward References

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

//Forward references for types whose
// size we don't need to know
class Department;
class Photo;

class Person {
public:
      //Class members, all types are passed by reference or pointer
      Person(const string& name, const string& address);
      void setDept(const Department& dept);
      Photo * buildPicture();
      void printDetails(const iostream& stream);
}
```

The Compiler Firewall Idiom

- Forward references can be used to remove all dependencies on a classes internals
  - Helpful for classes which are shared between different layers of your architecture and will introduce unwanted dependencies
  - This can considerably speed up your build times
- Your class has a single field
  - A pointer to a struct that contains the classes implementation
    - Usually this holds both the fields and the private methods
    - All references to fields in public methods are therefore indirect
  - Because it is a pointer the type doesn't need to be visible from the declaration of the main class

The Compiler Firewall Idiom

```
class StackImpl;

// All the private members of Stack
//  are encapsulated by StackImpl
class Stack {
private:
      StackImpl* pimpl;
public:
      Stack();
      ~Stack();
      void add(string item);
      int size();
      string removeLast();
      string get(int index);
};
```

```
class StackImpl {
private:
      Node *first;
      int currentSize;
public:
      StackImpl();
      void add(string item);
      int size();
      string removeLast();
      string get(int index);
};
class Node{
private:
      string item;
      Node *next;
public:
      Node(string in):item(in),next(0){}
      void setNext(Node* n=0){ next=n; }
      Node* getNext(){ return next; }
      string getItem(){ return item; }
};
```


Operator Overloading

- Most operators can be overloaded
  - As a method of the class
  - As a global function that is a friend of the class
- The second option is more flexible
  - For example if 'operator+' is a method then you can write 'obj1 + obj2' but not '7 + obj2'
- Stream operators need to be global functions
  - They should return a reference to the stream to allow chaining
  - If using streams for output other then debugging then you should take account of any formatting flags that have been set
    - For example 'setw(26)' or 'setprecision(5)'

Operator Overloading

```
class Number {
      friend Number operator+(const Number&, const Number&);
private:
      int value;
public:
      Number(int num);
      void print(ostream& output) const;
};
Number::Number(int num) {
      value = num;
}
void Number::print(ostream& output) const {
      output << "The value in this number is: " << value << endl;
}
Number operator+(const Number& lhs, const Number& rhs) {
      int total = lhs.value + rhs.value;
      Number tmp(total);
      return tmp;
}
```


Operator Overloading

```
ostream& operator<<(ostream& output, const Number& rhs) {
      rhs.print(output);
      return output;
}
int main() {
      Number no1(8);
      Number no2(2);

      Number no3 = no1 + no2;
      //prints The value in this number is: 10
      cout << no3;

      Number no4 = 12 + no3;
      //prints The value in this number is: 22
      cout << no4;
}
```


Other Types of Inheritance

- Public inheritance is widely used in C++
  - Protected and private inheritance are not (for good reason)
- Non-public inheritance qualifies the 'IS-A' relationship
  - Private inheritance means that only the derived class should know that it inherits from base
    - All inherited members are hidden from clients and subclasses
    - Only the derived class can cast a derived pointer to the base type
  - Protected inheritance means that only the derived class and its subclasses should know that it inherits from base
    - All inherited members are hidden, with the exception of subclasses
    - Only the derived class and subclasses can up-cast a derived pointer

Issues with Inheritance

- Inheritance is not the hold grail of OO
  - It quite clearly violates encapsulation
    - Unless you can resist making private fields protected
  - It complicates your class design
    - Creating classes that other developers can inherit from is difficult
    - Especially if you are developing frameworks for general use
  - It creates an unbreakable binary dependency
    - Your class cannot be used by clients without their compiler also seeing the full declaration of the base class
    - With composition this could have been hidden from the client
- C++ allows code that works against good inheritance
  - Such as the ability to hide a non-virtual base class function
  - Also the inclusion of private and protected kinds of inheritance

Issues with Inheritance

- A large inheritance hierarchy is hard to maintain
  - Developers lose track of where a member is declared
  - Inheritance trees should be bushy rather than deep
- There is a continual need to keep the tree balanced
  - Updates are frequently made to fix a problem in a leaf class
  - Normally this fix will need to be made in other leaf classes
  - If so there may be functionality that needs to be promoted
- One approach is to make all non-leaf classes abstract
  - Only the most derived classes can be instantiated
  - Instead of inheriting one concrete class from another we create a third abstract class that both the original classes can derive from
  - This clarifies what the shared abstraction and simplifies the code

Hiding Inherited Methods

- Overloading and overriding are often confused
  - Overloading is a convenience for developers
  - Overriding is implementing polymorphism via vtables
- Never override a non-virtual function
  - The method which gets called then depends on the type of the pointer, NOT the type of the object
  - This is the reverse of polymorphism and is horrible to debug
- You cannot add to the list of overloads in a derived class
  - Instead of adding an overload you hide the base class methods
  - You can avoid this via the using keyword e.g. ‘using base func;’

Hiding a Non-Virtual Base Function

```
int main() {
    Derived d;

    Base * bptr = &d;
    Derived * dptr = &d;

    Base & bref = d;
    Derived & dref = d;

    bptr->op(12);
    bref.op(12);

    dptr->op(12);
    dref.op(12);

    return 0;
}
```

```
class Base {
public:
   void op(int i) {
        cout << "Base::op called" << endl;
   }
};
class Derived : public Base {
public:
    void op(int i) {
        cout << "Derived::op called" << endl;
    }
};
```


Hiding Overloaded Base Methods

```
int main() {
    Derived d;
    d.op(5);

    return 0;
}
```

```
class Base {
public:
   void op(int i) {
        cout << "Base::op<int>" << endl;
   }
    void op(float f) {
        cout << "Base::op<float>" << endl;
    }
};
class Derived : public Base {
public:
    //this declaration hides the methods called 'op' declared
    // in class Base, it does NOT overload the method
    void op(double d) {
        cout << "Derived::op<double>" << endl;
    }
};
```


Pure Virtual Methods

- A pure virtual method is an empty slot
  - The entry is in the objects vtable but there is no matching functionality to be called
- The declaration is of form 'virtual void func() = 0'
  - Designed to indicate that the slot points nowhere
  - In the same way that a NULL pointer refers to nothing
  - Note it is platform dependant if 0 is an invalid location
    - So '= 0' does not literally mean 'points to zero'
- A pure virtual method can still have an definition
  - This can be used by derived classes as a partial implementation
  - Or it can be a full solution that derived classes must subscribe to

Interfaces

- An interface is a class that represents a contract
  - Sometimes the term Policy Class is used
    - Not to be confused with what the term means in templates
  - It specifies behaviour but contains no implementation
  - In C++ it is a class that only contains pure virtual methods
- Multiple inheritance is not disapproved of with interfaces
  - You can inherit from as many interfaces as you please
  - UML calls this realization to distinguish it from normal inheritance
- There are three key uses for interfaces
  - To define the client/server contract in DCOM/CORBA etc…
  - To cleanly separate different layers of the architecture
  - To speed up the build time by reducing dependencies

Interfaces

```
class ITransaction {
public:
    virtual bool addResource(const Resource& res) = 0;
    virtual bool removeResource(const Resource& res) = 0;
    virtual int getStatus() = 0;
    virtual void commit() = 0;
    virtual void abort = 0;
    virtual void rollback() = 0;
    virtual void setRollbackOnly() = 0;
};
```


Private Virtual Methods

- Private and virtual sounds like a contradiction
  - Normally we use virtual methods to provide slots for clients to call regardless of the type of the object
- A special case is when the base class is also the client
  - We can specify the algorithm inside the base class
    - For example make_connection -> send_data -> close_connection
  - The steps of the algorithm must be supplied by derived classes
    - So we provide a private virtual function for each step
    - Clients override these and fill in their own implementation

Private Virtual Methods

```
class BasicComms {
private:
        virtual void openConnection() = 0;
        virtual void transmit() = 0;
        virtual void closeConnection() = 0;
public:
        void sendMessage();
};

void BasicComms::sendMessage() {
        openConnection();
        transmit();
        closeConnection();
}
```

```
class SMTPComms : public BasicComms {
private:
        void openConnection();
        void transmit();
        void closeConnection();
};

void SMTPComms::openConnection() {
        cout << "Opening Connection" << endl;
}
void SMTPComms::transmit() {
        cout << "Transmitting Data" << endl;
}
void SMTPComms::closeConnection() {
        cout << "Closing Connection" << endl;
}
```


Methods and Default Parameters

- Default parameters are an alternative to overloading
  - Instead of 'void func(int)' and 'void func()' use 'void func(int i=0)'
  - Assuming there is a sensible default value to use
- Virtual methods should be wary of default values
  - Virtual method calls are dynamically bound whereas default values are statically bound
  - The definition to be called is worked out at runtime based on the type of the object but the default value is fixed at compile time
- If a derived class changes the value the result is strange
  - The value used depends on the type of the pointer or reference

Methods and Default Parameters

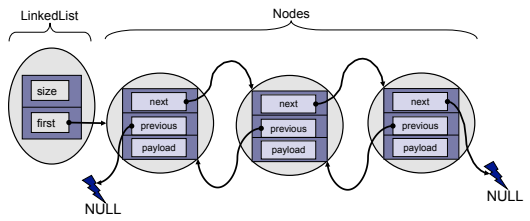
```
class Base {
public:
        virtual void func(int i=99) {
                cout << "Base::func called with value " << i << endl;
        }
};
class Derived : public Base {
public:
        //BAD - redefining default parameter value of virtual function
        void func(int i=100) {
                cout << "Derived::func called with value " << i << endl;
        }
};
int main() {
        Derived d;
        Derived * d_ptr = &d;
        Base * b_ptr = &d;

        d_ptr->func(); //prints Derived::func called with value 100
        b_ptr->func(); //prints Derived::func called with value 99
}
```

Project No 7

A Linked List Data Structure

garth@ggilmour.com

A Linked List Example



Exceptions

Error Handling in C++

garth@ggilmour.com

Exceptions

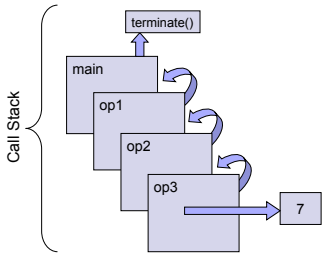
- C style error handling uses the return value
  - This limits the way you write your code and muddles business logic with error handling
- C++ attempts to solve this with exceptions
  - Exceptions are 'thrown' to handlers in another scope
  - This provides another way of exiting your function besides returning, exiting or crashing the program
- An exception can be almost anything
  - A basic type, pointer or object
  - Using objects as exceptions is recommended
  - For example the standard libraries use 'std::exception'

Throwing An Exception

- When an exception is thrown we check if it can be handled within the current function
- If this is not the case we unwind the stack
  - The value thrown is copied into a separate area of memory
  - We exit the current function and return to the calling function
  - If the invocation of the original function occurred inside an appropriate exception handler then we execute that handler
  - Otherwise the unwinding continues at the next level
- If the exception cannot be handled the program ends
  - The 'terminate' function is called which exits the program
  - It is possible to replace the terminate function

Unwinding the Call Stack

```
int main() {
    op1();
}
void op1() {
    op2();
}
void op2() {
    op3();
}
void op3() {
    if(x > 12) {
        throw 7
    }
}
```



Exceptions and Objects

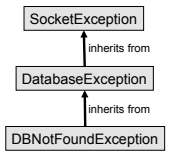
- Exceptions significantly impact class design
  - It adds a new section to the contract a method has with its client
- As the stack is unwound storage is de-allocated
  - The destructors of local objects will be called
  - A destructor should not throw a second exception
    - Because this will result in a call to the 'terminate' function
    - There can only be one exception on the go at any time
  - Only fully constructed objects have their destructors called
    - Partially created objects will not have a chance to release resources
    - This is a good reason to make your constructors exception safe

Handling Exceptions

- An exception handler consists of a try/catch block
  - The try block contains the code that may throw
  - The catch block takes a parameter that matches the exception
    - When the catch block is called the value thrown will be copied in
    - The exception may be re-thrown via the syntax 'throw;'
- Multiple catch blocks can be associated with a try
  - Catch blocks are examined in the order they were declared
  - The first block whose parameter is a viable match to the exception will be used, even if better matches were available
    - This is why catch blocks for a derived class are always placed before catch blocks for the corresponding base class
  - If no matching catch block can be found the stack is unwound

Handling Exceptions

```
try {
    db.connect("192.168.1.36","admin","passwd");
} catch (SecurityException & ex) {
    //handle exception here
} catch (DBNotFoundException & ex) {
    //handle exception here
} catch (DatabaseException & ex) {
    //handle exception here
} catch (SocketException & ex) {
    //handle exception here
}
```



Exception Specifications

- An exception specification lists the exception types that might be thrown from the function
  - Either directly or as a result of a stack unwind
  - For example 'int func() throw (int,double,MyClass)'
  - An empty specification signifies that the function will not throw
- An exception specification guarantees nothing
  - It is NOT a compile error for a function to call other functions that may throw exceptions of types not listed in its specification
    - E.g. 'void op1() throw(int)' may call 'void op2() throw MyClass'
  - If at runtime an exception is thrown which is not listed in a function exception specification then 'unexpected' is called
    - By default this will call 'terminate'

Declaring Templates

Generic Programming in C++

garth@ggilmour.com

Generic Programming

- Object Oriented development uses inheritance and polymorphism for related types
  - The classes are specializations of some abstraction and use polymorphism to redefine inherited behaviour
- Generic programming is for unrelated types
  - Where there is some algorithm whose structure remains the same regardless of the types used
  - The most obvious example is implementing collections of types
- C++ uses templates to implement generic code
  - You write the algorithm using placeholders for the types
  - The compiler creates type specific instances of your code

Creating Templates

- The code below would work for any type which:
  - Can have instances created initialized to zero
  - Can be assigned to an instance of itself
  - Supports the addition of two instances
  - Can be returned by value

```
double addAll(double * data_ptr, int length) {
      double total = 0;
      for(int i=0;i<length;i++) {
            total = total + data_ptr[i];
      }
      return total;
}
```


Creating Templates

- Templates can either be functions or classes
  - In each case we prefix the declaration with 'template < … >'
- Inside the braces we list our placeholders
  - For example 'template < typename T, typename U >'
  - Or alternatively 'template < class T, class U>'
- The convention is to use upper case letters
  - Starting with T and continuing with U etc…
- Either 'typename' or 'class' can introduce a placeholder
  - The former was introduced because 'class' is misleading
  - The placeholder can be of any type and not just class types

Creating Templates

```
template <typename T> T addAll(T * data_ptr, int length) {
      T total = 0;
      for(int i=0;i<length;i++) {
            total = total + data_ptr[i];
      }
      return total;
}
int main() {
      double darray[10] = {1,2,3,4,5,6,7,8,9,10};   //Template instance for double
      double resultOne = addAll(darray,10);
      std::cout << "First total is: " << resultOne << std::endl;

      int iarray[10] = {1,2,3,4,5,6,7,8,9,10};          //Template instance for integer
      int resultTwo = addAll(iarray,10);
      std::cout << "Second total is: " << resultTwo << std::endl;

      Number numbers[] = {1,2,3,4,5,6,7,8,9,10};   //Template instance for Number
      Number resultThree = addAll(numbers,10);
      std::cout << "Third total is: " << resultThree << std::endl;
}
```

Creating Templates

- Function templates have two kinds of parameter
  - The type placeholders are referred to as type parameters
  - The actual arguments are call parameters
- Call parameters can be used to identify type parameters
  - This is called function template argument deduction
- The type parameters can be specified explicitly
  - For example 'addAll<short>(sarray,10)'
  - This is required if there is a choice between two or more types
    - The compiler does not automatically convert to the larger type

Template Errors

- Generic code can produce ambiguities
  - Where a C++ expression has multiple interpretations
- For example consider 'T::Mystery * p'
  - 'Mystery' could be either a nested type or static field
  - If the former then the expression declares a pointer called 'p'
  - If the latter then we want to multiply the static field by 'p'
- The 'typename' keyword resolves this confusion
  - 'typename T::Mystery' specifies that 'Mystery' is a type
- Templates can also have problems with inheritance
  - The compiler may be unable to see that a symbol is inherited
  - This can be resolved by qualifying the symbol with 'this->'

Template Errors

- Templates cause problems for linkers
  - When the compiler sees 'addAll(iarray,10)' it generates a symbol that must be linked against a template instance for integers
  - But if the '.cpp' file containing the template has already been compiled the instance has probably not been created
- There are two approaches to this problem
  - Declare and define your template entirely in a header file
    - This is known as 'Inclusion Model' and is the most common solution
  - Explicitly create instances of your template in its '.cpp' file
    - This is called explicit instantiation e.g. 'template int addAll(int,int)'
    - This avoids 'header bloat' but means that you must manually keep track of each template instance that your program needs to create

Nontype Template Parameters

- Template parameters can be constant values
  - For example 'template<typename T, int SIZE>'
  - These are nontype template parameters
- These can be used to hard code values into one particular instance of the template
  - For example setting the default size of a collection
- To simplify template creation the range of types that can be used for nontype template parameters is limited
  - Integer values and pointer to external objects are OK
  - Floating point values and objects are not allowed

Nontype Template Parameters

```
template <typename T, int SIZE> T addAll(T * data_ptr) {
	T total = 0;
	for(int i=0;i<SIZE;i++) {
		total = total + data_ptr[i];
	}
	return total;
}
int main() {
	//Create two instances of the template
	double(* d_ptr)(double *) = addAll<double,10>;
	Number(* n_ptr)(Number *) = addAll<Number,10>;

	double darray[10] = {1,2,3,4,5,6,7,8,9,10};
	double resultOne = d_ptr(darray);
	std::cout << "First total is: " << resultOne << std::endl;

	Number numbers[] = {1,2,3,4,5,6,7,8,9,10};
	Number resultTwo = n_ptr(numbers);
	std::cout << "Second total is: " << resultTwo << std::endl;
}
```


Multiple Template Parameters

- Complex templates frequently have multiple types
  - For example a type 'T' for the elements in the container and a type 'U' for a iterator that can be used to navigate them
- Even with multiple template parameters generic code can work for both built in and user defined types
  - In some containers 'U' may just be a pointer whereas in others it may be a built in type designed to navigate the container
- The compiler only requires that the type has methods and operators that match those used in the template
  - Operator overloading lets the user defined types be manipulated as if they were built in types

Multiple Template Parameters

```
class Size {
      friend operator<(Size& lhs, Size& rhs);
private:
      int size;
public:
      Size(int i);
      operator int();
      Size& operator++(int);
};
operator<(Size& lhs, Size& rhs) {
      return lhs.size < rhs.size;
}
Size::Size(int i) : size(i) {
}
Size& Size::operator++(int) {
      size++;
      return *this;
}
Size::operator int() {
      return size;
}
```


Multiple Template Parameters

```
template <typename T, typename U> T addAll(T * data_ptr, U length) {
      T total = 0;
      for(U i=0;i<length;i++) {
            total = total + data_ptr[i];
      }
      return total;
}

int main() {
      //Template instance for double
      double darray[10] = {1,2,3,4,5,6,7,8,9,10};
      double resultOne = addAll(darray,10);
      std::cout << "First total is: " << resultOne << std::endl;

      //Template instance for Number and Size
      Number numbers[] = {1,2,3,4,5,6,7,8,9,10};
      Size arraySize(10);
      Number resultTwo = addAll(numbers,arraySize);
      std::cout << "Second total is: " << resultTwo << std::endl;
}
```


Specializing Templates

- Function templates can be overloaded
  - The rules are the same as for normal templates
- Both class and function templates can be specialized
  - Specialization is used when an algorithm is only partially generic
  - The algorithm works fine in the majority of cases but for certain template parameters an adjusted version is required
    - For example where user defined types don't support certain operators or can be processed more efficiently via a special idiom
- Templates can be fully or partially specialized
  - A full specialization replaces all the template parameters with named types, for example 'template<> int addAll(int,int)'
  - A partial specialization affects a subset of the parameters, for example 'template<typename U> int addAll(int,U)'

Specializing Templates

```
//The basic template
template <typename T, typename U> T addAll(T * data_ptr, U length) {
	T total = 0;
	for(U i=0;i<length;i++) {
		total = total + data_ptr[i];
	}
	return total;
}

//Define a partial specialization for Size objects
template <typename T> T addAll(T * data_ptr, Size& size) {
	std::cout << "Partial specialization for Size used" << std::endl;
	T total = 0;
	for(int i=0;i<size.getValue();i++) {
		total = total + data_ptr[i];
	}
	return total;
}
```


Specializing Templates

```
int main() {
	//Template instance for double
	double darray[10] = {1,2,3,4,5,6,7,8,9,10};
	double resultOne = addAll(darray,10);
	std::cout << "First total is: " << resultOne << std::endl;

	//Template instance for integer
	int iarray[10] = {1,2,3,4,5,6,7,8,9,10};
	int resultTwo = addAll(iarray,10);
	std::cout << "Second total is: " << resultTwo << std::endl;

	//Tempate instance for Number and Size
	Number numbers[] = {1,2,3,4,5,6,7,8,9,10};
	Size arraySize(10);
	Number resultThree = addAll(numbers,arraySize);
	std::cout << "Third total is: " << resultThree << std::endl;
}
```


Template Template Parameters

- Templates are useful in isolation
  - But at their most powerful when combined
- Consider the design of a collections library
  - Class templates can be designed for data structures such as arrays, lists, trees and hashes
  - Function templates can be written for searching and sorting which work across all data structures
  - As used in the Standard Template Library (STL)
- Templates composed of templates have one problem
  - Names of symbols become very large and hard to debug
  - Pragmas can be used to prevent warnings about symbol names

Template Template Parameters

```
unresolved external symbol "void __cdecl showContents<class std::list<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char>
>,class std::allocator<class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > > >(class std::list<class
std::basic_string<char,struct std::char_traits<char>,class std::allocator<char>
>,class std::allocator<class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> > > > &,class
std::basic_ostream<char,struct std::char_traits<char> > &)"
```

```
error C2784: 'std::basic_string<_Elem,_Traits,_Alloc> std::operator +(const
_Elem *,const std::basic_string<_Elem,_Traits,_Alloc> &)' : could not deduce
template argument for 'const T1 *' from 'std::list<_Ty>::iterator'  with
[          _Ty=std::string ]
```


Template Template Parameters

```
//The basic template
template <typename T, typename U> T addAll(T * data_ptr, U length) {
      T total = 0;
      for(U i=0;i<length;i++) {  total = total + data_ptr[i];  }
      return total;
}
//Define a specialization for Numbers
template <> Number addAll(Number * data_ptr, int length) {
      cout << "Specialization for number used" << endl;
      Number total = 0;
      for(int i=0;i<length;i++) {  total = total + data_ptr[i];  }
      return total;
}
//Define a specialization for vectors - note that typename is required for second parameter
// because the compiler needs to be explicitly told that iterator is a dependant type
template <typename T> T addAll(vector<T> * data, typename vector<T>::iterator length) {
      cout << "Specialization for vector used" << endl;
      T total = 0;
      for(vector<T>::iterator i = data->begin(); i<length; i++) {  total = total + *i;  }
      return total;
}
```


Template Template Parameters

```
int main() {
      //Template instance for double
      double darray[10] = {1,2,3,4,5,6,7,8,9,10};
      double resultOne = addAll(darray,10);
      std::cout << "First total is: " << resultOne << std::endl;

      //Template instance for integer
      int iarray[10] = {1,2,3,4,5,6,7,8,9,10};
      int resultTwo = addAll(iarray,10);
      std::cout << "Second total is: " << resultTwo << std::endl;

      //Template specialization for Number
      Number numbers[] = {1,2,3,4,5,6,7,8,9,10};
      Number resultThree = addAll(numbers,10);
      std::cout << "Third total is: " << resultThree << std::endl;
```

Template Template Parameters

```
    //Template specialization for vector<T>
    vector<int> intVector;
    intVector.push_back(1);
    intVector.push_back(2);
    intVector.push_back(3);
    intVector.push_back(4);
    intVector.push_back(5);
    intVector.push_back(6);
    intVector.push_back(7);
    intVector.push_back(8);
    intVector.push_back(9);
    intVector.push_back(10);
    int resultFour = addAll(&intVector,intVector.end());
    std::cout << "Fourth total is: " << resultFour << std::endl;
}
```


Class Templates

- Class templates are similar to function templates
  - The declaration is prefixed with the type parameters
  - The type parameters can be used in the declaration of fields, parameters and local variables
- Method definitions need the full type qualification
  - For example 'List<T>::add(T& item)'
- Class templates can contain other templates
  - Methods can themselves be function templates
  - Nested classes can also be class templates
  - If a method is a function template it cannot be virtual
  - Other templates can be friends of the current class template

List Template Pt1: The Node Class

```
template<typename T> class Node {
public:
    Node(T item) ;
    Node(T item, Node<T> * next, Node<T> * prev);
    Node<T> * getNext();
    void setNext(Node<T> * next);
    Node<T> * getPrev();
    void setPrev(Node<T> * prev);
    T getItem();
private:
    Node<T> * next;
    Node<T> * prev;
    T item;
};
```

```
template<typename T> Node<T>::Node(T item) {
	this->item = item;
	next = prev = 0;
}
template<typename T> Node<T>::Node(T item, Node<T> * next,
		Node<T> * prev) {
	this->item = item;
	this->next = next;
	this->prev = prev;
}
template<typename T> Node<T> * Node<T>::getNext() {
	return next;
}
template<typename T> void Node<T>::setNext(Node<T> * next) {
	this->next = next;
}
template<typename T> Node<T> * Node<T>::getPrev() {
	return prev;
}
template<typename T> void Node<T>::setPrev(Node<T> * prev) {
	this->prev = prev;
}
template<typename T> T Node<T>::getItem() {
	return item;
}
```


List Template Pt2: The List Class

```
template<typename T> class List {
public:
	List();
	~List();
	void add(T item);
	void print();
	void remove(int index);
	T get(int index);
private:
	int size;
	Node<T> * first;
};
```



```
template<typename T> List<T>::List() : size(0), first(0) {
}
template<typename T> List<T>::~List() {
	Node<T> * toDie = first;
	while(toDie) {
		Node<T> * tmp = toDie;
		toDie = toDie->getNext();
		delete tmp;
	}
}
template<typename T> void List<T>::add(T item) {
	if(size == 0) {
		first = new Node<T>(item);
	} else {
		Node<T> * tmp = first;
		while(tmp->getNext() != 0) {
			tmp = tmp->getNext();
		}
		tmp->setNext(new Node<T>(item,0,tmp));
	}
	size++;
}
```

```
template<typename T> void List<T>::print() {
	if(size == 0) {
		cout << "list is empty" << endl;
	} else {
		cout << "contents are:" << endl;
		Node<T> * tmp = first;
		while(tmp) {
			cout << tmp->getItem() << endl;
			tmp = tmp->getNext();
		}
	}
}
template<typename T> T List<T>::get(int index) {
	Node<T> * tmp = first;
	for(int i=0;i<index;i++) {
		tmp = tmp->getNext();
	}
	return tmp->getItem();
}
```



```
template<typename T> void List<T>::remove(int index)
{
	Node<T> * tmp = first;
	if(index == 0) {
		first = first->getNext();
		if(first != 0) {
			first->setPrev(0);
		}
	} else {
		for(int i=0;i<index;i++) {
			tmp = tmp->getNext();
		}
		if(tmp->getNext() == 0) {
			tmp->getPrev()->setNext(0);
		} else {
			Node<T> * before = tmp->getPrev();
			Node<T> * after = tmp->getNext();
			before->setNext(after);
			after->setPrev(before);
		}
	}
	delete tmp;
	size--;
}
```


List Template Pt3: The Client Class

```
int main() {
	List<string> list;
	cout << "Empty list" << endl;
	list.print();

	list.add("abc");
	list.add("def");
	list.add("ghi");
	list.add("jkl");
	list.add("mno");
	list.add("pqr");
	list.add("stu");

	cout << "Full list" << endl;
	list.print();
```

```
	list.remove(0);
	cout << "List with 0 removed" << endl;
	list.print();

	list.remove(5);
	cout << "List with 5 removed" << endl;
	list.print();

	list.remove(2);
	cout << "List with 2 removed" << endl;
	list.print();

	cout << "Final list contents" << endl;
	cout << "\t" << list.get(0) << endl;
	cout << "\t" << list.get(1) << endl;
	cout << "\t" << list.get(2) << endl;
	cout << "\t" << list.get(3) << endl;

	cout << "End of Program" << endl;
}
```

Traits Classes

- Complex templates rely on two features
  - Specialization acts like a compile time switch statement
    - The compiler works out what type the client has passed and uses the appropriate specialization
  - Templates are compiled conditionally
    - If part of a template is not required then it will not be compiled
- These features can be combined to work out characteristics of a type at compile time
  - Similar to the runtime functionality provided by the introspection and reflection libraries in Java and C#

A Traits Class

```
template <typename T>
class Traits {
private:
	template <typename U> struct PTraits {
		enum { isPointer = false, isReference = false };
	};
	template <typename U> struct PTraits<U*> {
		enum { isPointer = true, isReference = false };
	};
	template <typename U> struct PTraits<U&> {
		enum { isPointer = false, isReference = true };
	};
public:
	enum { isPointer = PTraits<T>::isPointer,
		isReference = PTraits<T>::isReference };
};
```


A Traits Class

```
void main() {

	cout << Traits<int>::isPointer << endl;
	cout << Traits<int *>::isPointer << endl;
	cout << Traits<int &>::isPointer << endl;
	cout << Traits<int>::isReference << endl;
	cout << Traits<int *>::isReference << endl;
	cout << Traits<int &>::isReference << endl;
}
```

Using the STL

Containers, Iterators and Algorithms

garth@ggilmour.com

The Standard Template Library

- The STL has three types of component
  - Containers, Iterators and Algorithms
- A container holds arbitrary values
  - Sequential containers hold an ordered collection of values
  - Associative containers hold values indexed by keys
- Iterators allow access to a container
  - Without being reliant on its underlying type
  - An iterator may allow forward, bidirectional or random access
- Algorithms implement generic operations
  - Which are not dependant on the type in the container
  - For example searching, sorting and splicing

Using the STL

- You should prefer the STL over 3rd party libraries
  - Unless your application is built around MFC, QT etc…
- The STL has a massive feature set
  - Always check a good STL guide before reinventing the wheel
- Don't put 'using namespace std' in header files
  - In case another project uses your header and has name clashes
  - This is quite likely given the number of types declared in std
  - Prefer to bring in each type explicitly e.g. 'using std::string'
    - This is a good disciple for keeping header files small
- In cpp files put 'using namespace std' after all includes
  - To prevent hard to debug name clashes in 3rd party headers

Iterators

- An iterator is an entity that can traverse a container
  - It supports the pointer syntax for moving and reading values
  - E.g '*iter' gets the value 'iter' is pointing to while 'iter++' moves it
- There are two template functions for iterators
  - 'distance' find the distance between two iterators
  - 'advance' moves an iterator a number of elements
- There are different types of iterator
  - An input iterator reads while an output iterator writes
  - An iterator may be forward, reverse or bidirectional
  - A random access iterator is a bidirectional iterator which overloads the '[ ]' operator to jump to any value

Iterators

- Iterators encapsulate most access to containers
  - The STL algorithms take iterators to delimit sequence of values
- Insert operators allow values to be added to a sequence
  - They insert at the front, the back or an arbitrary position
- Iterators are usually created by the container
  - A 'begin' method provides an iterator to the first element and an 'end' method to after the last
  - Most algorithms return an iterator pointing to the end of the sequence to signal that that operation has failed
- For many containers an iterator will just be a pointer
  - Your code should never assume this to remain portable


```
int main() {
	vector<string> vec;

	vec.push_back("aaa");
	vec.push_back("bbb");
	vec.push_back("ccc");
	vec.push_back("ddd");
	vec.push_back("eee");
	vec.push_back("fff");
	vec.push_back("ggg");

	vector<string>::iterator iterOne = vec.begin();
	vector<string>::iterator iterTwo = vec.end();

	iterTwo--;
	long length = distance(iterOne,iterTwo);
	cout << "Values of iterators are: " << *iterOne << " and " << *iterTwo << endl;
	cout << "Initial space between iterators is: " << length << endl;

	advance(iterOne,2);
	advance(iterTwo,-2);
	length = distance(iterOne,iterTwo);
	cout << "New values of iterators are: " << *iterOne << " and " << *iterTwo << endl;
	cout << "New space between iterators is: " << length << endl;
}
```

The STL Containers

- There are four sequential and four associative containers
  - Vector, string, deque and list are sequential
  - Set, multiset, map and multimap are associative
- Some containers are implemented in terms of continuous blocks of memory
  - Vector, string and deque
- Others are implemented as nodes connected by pointers
  - List and all the associative containers
- Choosing the correct container is important
  - Both for speed and for ease of programming

The Vector Container

- Vector is now the standard C++ container
  - You should always prefer vectors to arrays
  - Your code is safer and you can use algorithms
- A vector is essentially a managed array
  - Its elements are guaranteed to be contiguous
  - This means that you can pass the address of vector elements into C code e.g. '&vec[10]'
- Vectors support random access
  - Via the 'push_back' and 'insert' methods
- There is one template specialization for 'vector<bool>'
  - You need to be careful about using this

The Vector Container

```
void printUsingAccess(vector<string> vec) {
	for(unsigned int i=0; i<vec.size(); i++) {
		cout << "Element at index " << i << " is " << vec[i] << endl;
	}
	cout << endl;
}
void printUsingIterators(vector<string> vec) {
	vector<string>::iterator iter = vec.begin();
	cout << "Contents are: ";
	for(iter;iter != vec.end();iter++) {
		cout << *iter << " ";
	}
	cout << endl << endl;
}
```

The Vector Container

```
int main() {
	vector<string> vec;

	string sarray[] = {"abc","def","ghi","jkl","mno","pqr","stu","vwx","yza"};
	for(int i=0; i<9; i++) {
		vec.push_back(sarray[i]);
	}

	printUsingAccess(vec);
	printUsingIterators(vec);

	vector<string>::iterator iter = vec.begin() += 2;
	vec.insert(iter,2,"XXXXXXX");

	printUsingIterators(vec);
}
```


The List Container

- A list is like a vector but written as a linked list of nodes
  - The elements are not arranged contiguously
  - By default you should choose vector over list
- The list container has one major feature
  - Modifying the middle of the list does not mean that the following elements need to be moved
    - Instead we only need to snip some pointers
  - Insert and erase are constant time operations
- Otherwise lists have several disadvantages
  - Legacy code cannot treat them as arrays
  - Random access is not supported

The List Container

```
int main() {
	list<string> myList;

	string sarrayOne[] = {"abc","def","ghi","jkl"};
	for(int i=0; i<4; i++) {
		myList.push_back(sarrayOne[i]);
	}
	string sarrayTwo[] = {"www","xxx","yyy","zzz"};
	for(int i=0; i<4; i++) {
		myList.push_front(sarrayTwo[i]);
	}
	printUsingIterators(myList);

	list<string>::iterator iter = myList.begin();
	iter++;
	myList.insert(iter,2,"XXXXXXX");
	printUsingIterators(myList);
}
```

The Deque Container

- A deque is a double ended queue
  - It can be visualized as three memory blocks representing the start, middle and end of the sequence
- It has some special characteristics
  - Random access is supported (as with vector)
  - Insertion and removal at an end takes constant time (as with list)
  - Pointers or references to the middle of the deque are not invalidated by insertions at the ends
- Deque combines the strengths of vector and list
  - But vector should always be your default choice

The Associative Containers

- A ‘set’ is technically an associative container
  - Even though the key is also the value
  - The ‘multiset’ container allows identical values
- A ‘map’ stores key/value pairs
  - The ‘multimap’ allows multiple entries to have the same key
  - Changing a keys value could break the sorting order
- The ‘pair’ struct represents entries inside the map
  - It is a simple template class defined in ‘<utility>’
  - If you iterate over a map a ‘pair’ object is returned when you dereference the iterator

The Map Container

```
void printMap(map<int,string> container) {
    map<int,string>::iterator iterator;

    for(iterator = container.begin(); iterator != container.end(); iterator++) {
        pair<int,string> p = *iterator;
        cout << p.first << " " << p.second << endl;
    }
    cout << endl;
}
void populateMap(map<int,string>& m) {
    typedef map<int,string>::value_type entry;

    string sarray[] = {"abc","def","ghi","jkl","mno","pqr","stu","vwx","yza"};
    int keys[] = {1,2,3,4,5,6,7,8,9};
    for(int i=0; i<9; i++) {
        m.insert(entry(keys[i], sarray[i]));
    }
}
```

The Map Container

```
int main() {
    map<int,string> tst_map;

    populateMap(tst_map);
    printMap(tst_map);

    map<int,string>::iterator iterator = tst_map.find(5);

    cout << "Key 5 associated with string: " << iterator->second << endl;
    cout << "Key 8 associated with string: " << tst_map[8] << endl << endl;

    tst_map.erase(3);
    tst_map.erase(6);
    printMap(tst_map);
}
```


Using Containers

- Containers hold copies of the values you add to them
  - The copy constructor and copy assignment operator are used as the value is moved about
  - If you implement your own they should be efficient
- One option is to hold pointers in the container
  - If these point to heap memory you must delete them manually
  - This can cause problems when using algorithms
- Containers of type 'auto_ptr' are illegal
  - When the 'auto_ptr' is copied the copy takes over ownership
  - This means that otherwise innocuous container operations can have dangerous side effects

The Dimensions of a Container

- The number of elements in a container and its total size are not necessarily the same thing
  - 'size' returns the number of elements
  - 'capacity' returns the number that can be held
  - 'resize' changes the number of elements
  - 'reserve' changes the containers capacity
- With 'vector' and 'string' you should reserve a size
  - Appropriate to the number of elements you might hold
  - This will avoid reallocations and unnecessary copying
- Use 'myList.empty()' in preference to 'myList.size() == 0'
  - The implementation of size may try to count all the nodes

The STL Algorithms

- STL algorithms can be categorized as:
  - Those which use a container without modifying it
  - Those which change the order of elements in a container
  - Those which change the contents of the container
  - Those which change the elements in the container
- Some algorithms need to work with elements
  - This is achieved via function pointers or functors
  - A functor is an object which overloads 'operator()'
- Sometimes an algorithm duplicates a container method
  - If in doubt prefer to use the method offered by the container

The Non Mutating STL Algorithms

Algorithm	Description
for_each	Performs an operation on each element
find	Finds the position of an element matching a value
find_if	Finds the position of an element matching a predicate
find_end	Finds the position of a subsequence
find_first_of	Finds the first occurrence of a member of another sequence
adjacent_find	Finds adjacent elements that match a value or predicate
count	Counts the number of elements matching a value
count_if	Counts the number of elements matching a predicate
mismatch	Checks if two containers have identical content
equal	Checks if two containers have identical content
search	Searches for a subsequence
search_n	Searches for a subsequence of N identical elements

The For Each Algorithm

- This is one of the most popular algorithms
  - It removes the need to manually loop over collections
- It performs an operation on each element
  - The operation can be either a free function or a functor
  - Any return value from the function is ignored
- For simple tasks use a free function
  - Such as printing the elements to a stream
- If you need to store data between calls use a functor
  - Its fields can accumulate information as it is reused

The For Each Algorithm

```
class PrintFunctor {
      ostream& output;
public:
      PrintFunctor(ostream& output) : output(output){}
      template<typename T> void operator()(T& value) {
            output << value << endl;
      }
};

template<typename T> void showContents(T& container, ostream& output) {
      T::iterator start = container.begin();
      T::iterator stop = container.end();
      PrintFunctor functor(output);

      for_each(start,stop, functor);
}
```


The For Each Algorithm

```
int main() {
      vector<string> vec;
      vec.push_back("abc");
      vec.push_back("def");
      vec.push_back("ghi");
      vec.push_back("jkl");

      list<int> lst;
      lst.push_back(20);
      lst.push_back(30);
      lst.push_back(40);
      lst.push_back(50);

      cout << "Vector contents are: " << endl;
      showContents(vec,cout);
      cout << endl << "List contents are: " << endl;
      showContents(lst,cout);
}
```


The Find Algorithms

- The find algorithms search for values
  - They return an iterator to where the value is located
  - If unsuccessful the iterator points to the sequence end
- There are 5 different find algorithms
  - ‘find’ searches for a value that you supply
  - ‘find_if’ uses a predicate to select the value
  - ‘find_end’ searches a sequence for a subsequence
  - ‘find_first_of’ searches the sequence for the first occurrence of any value found in a second sequence
  - ‘adjacent_find’ searches for two adjacent identical elements

The Find Algorithms

```
int main() {
    vector<string> vec;
    vec.push_back("abc");
    vec.push_back("aaa");
    vec.push_back("def");
    vec.push_back("def");
    vec.push_back("ghi");
    vec.push_back("aaa");
    vec.push_back("jkl");
    vec.push_back("mno");
    vec.push_back("pqr");
    vec.push_back("aaa");
    vec.push_back("stu");
    vec.push_back("vwx");
    vec.push_back("vwx");
    vec.push_back("yza");
    vec.push_back("aaa");
    vec.push_back("abc");

    showFindAlgorithm(vec);
    showFindIfAlgorithm(vec);
    showFindEndAlgorithm(vec);
    showFindFirstOfAlgorithm(vec);
    showAdjacentFindAlgorithm(vec);
}
```


The Find Algorithm

```
void showFindAlgorithm(vector<string>& vec) {
    vector<string>::iterator end = vec.end();
    vector<string>::iterator start = vec.begin();
    vector<string>::iterator current = start;

    while(1) {
        current = find(start,end,"aaa");
        if(current != end) {
            cout << "Found search string at position: "
                 << current - vec.begin() << endl;
            start = ++current;
        } else {
            break;
        }
    }
}
```


The Find If Algorithm

```
void showFindIfAlgorithm(vector<string>& vec) {
    vector<string>::iterator end = vec.end();
    vector<string>::iterator start = vec.begin();
    vector<string>::iterator current = start;
    StartsWithFunctor sf;

    while(1) {
        current = find_if(start,end,sf);
        if(current != end) {
            cout << "String: " << *current
                 << " at " << current - vec.begin()
                 << " starts with a" << endl;
            start = ++current;
        } else {
            break;
        }
    }
}
```

```
class StartsWithFunctor {
public:
    bool operator()(string& value) {
        return value[0] == 'a';
    }
};
```

The Find End Algorithm

```
void showFindEndAlgorithm(vector<string>& vec) {
    vector<string> target;
    target.push_back("ghi");
    target.push_back("aaa");
    target.push_back("jkl");
    target.push_back("mno");

    vector<string>::iterator position;

    position = find_end(vec.begin(),vec.end(),target.begin(),target.end());

    if(position == vec.end()) {
        cout << "The sub-sequence cannot be found" << endl;
    } else {
        cout << "The sub-sequence begins at position "
             << position - vec.begin() << endl;
    }
}
```


The Find First Of Algorithm

```
void showFindFirstOfAlgorithm(vector<string>& vec) {
    vector<string> target;
    target.push_back("ghi");
    target.push_back("aaa");
    target.push_back("jkl");
    target.push_back("mno");

    vector<string>::iterator position;
    position = find_first_of(vec.begin(),vec.end(),target.begin(),target.end());

    if(position == vec.end()) {
        cout << "None of the sub-sequence elements
                 are present in the main sequence" << endl;
    } else {
        cout << "The first member of the sub-sequence
                 to be found in the main sequence is "
             << *position << " at position " << position - vec.begin() << endl;
    }
}
```


The Adjacent Find Algorithm

```
class StartsNotWithFunctor {
public:
    bool operator()(string& value1,string& value2) {
        return (value1 == value2) && (value1[0] != 'd');
    }
};

void showAdjacentFindAlgorithm(vector<string>& vec) {
    cout << endl << endl << "showFindFirstOfAlgorithm" << endl;

    vector<string>::iterator position;

    position = adjacent_find(vec.begin(),vec.end());

    if(position != vec.end()) {
        cout << "First pair of adjacent and equal elements begins at "
             << position - vec.begin() << endl;
    }
```

The Adjacent Find Algorithm

```
    StartsNotWithFunctor sf;
    position = adjacent_find(vec.begin(),vec.end(),sf);

    if(position != vec.end()) {
        cout << "First pair of adjacent and equal elements not starting with d begins at "
             << position - vec.begin() << endl;
    }
}
```


Program Output Part I

```
showFindAlgorithm
Found search string at position: 1
Found search string at position: 5
Found search string at position: 9
Found search string at position: 14

showFindIfAlgorithm
String: abc at 0 starts with a
String: aaa at 1 starts with a
String: aaa at 5 starts with a
String: aaa at 9 starts with a
String: aaa at 14 starts with a
String: abc at 15 starts with a
```


Program Output Part II

```
showFindEndAlgorithm
The sub-sequence begins at position 4

showFindFirstOfAlgorithm
The first member of the sub-sequence to be found in the main sequence is aaa at
position 1

showFindFirstOfAlgorithm
First pair of adjacent and equal elements begins at 2
First pair of adjacent and equal elements not starting with d begins at 11
Press any key to continue
```

The Count Algorithm

- This counts the occurrences of a value in a sequence
  - Originally the count was passed as a reference parameter
  - In the current version of the STL it is returned from the function
- The 'count_if' algorithm uses a predicate
  - To decide whether an element should be counted
  - This is a simpler than using 'for_each' to count elements
- Both count and find can test if an element exists
  - Writing 'if(count(…))' is simpler than 'if(find(…) != v.end())'
  - But the latter is more efficient as find stops when it finds the element whereas count always continues to the end

The Count Algorithm

```
int main() {
	vector<string> vec;
	vec.push_back("zzz");
	vec.push_back("aaaaa");
	vec.push_back("aaaa");
	vec.push_back("aaa");
	vec.push_back("zzz");
	vec.push_back("bbbbb");
	vec.push_back("bbbb");
	vec.push_back("bbb");
	vec.push_back("zzz");

	int number = count(vec.begin(),vec.end(),"zzz");
	cout << "There are " << number << " elements with value zzz" << endl;

	StringLengthFunctor slf;
	number = count_if(vec.begin(),vec.end(),slf);
	cout << "There are " << number << " elements with five characters" << endl;
}
```

```
class StringLengthFunctor {
public:
	bool operator()(string& value) {
		return value.length() == 5;
	}
};
```


The Equal & Mismatch Algorithms

- The equal algorithm returns a boolean to show whether two sequences have identical content
  - An overloaded version takes a predicate to determine what identical means for your objects
- The mismatch algorithm is similar to equal
  - Except that it returns a pair of iterators
  - If the sequences are identical the first iterator in the pair refers to the end of the first sequence
  - If the sequences are not identical the iterators refer to where the mismatch occurs in the corresponding container
  - Again there is an overloaded version which takes a predicate

The Equal Algorithm

```
int main() {
	vector<string> vecOne;
	vecOne.push_back("abc");
	vecOne.push_back("def");
	vecOne.push_back("ghi");
	vecOne.push_back("jkl");
	vecOne.push_back("mno");

	vector<string> vecTwo;
	vecTwo.push_back("abc");
	vecTwo.push_back("def");
	vecTwo.push_back("ghi");
	vecTwo.push_back("jkl");
	vecTwo.push_back("mno");
```

```
	vector<string> vecThree;
	vecThree.push_back("abc");
	vecThree.push_back("def");
	vecThree.push_back("ghi");
	vecThree.push_back("zzz");
	vecThree.push_back("jkl");
	vecThree.push_back("mno");

	showEqualAlgotithm(vecOne,vecTwo);
	showEqualAlgotithm(vecOne,vecThree);
}
```


The Equal Algorithm

```
void showEqualAlgotithm(vector<string> p1,vector<string> p2) {

	bool result = equal(p1.begin(),p1.end(),p2.begin());

	if(result) {
		cout << "Containers match" << endl;
	} else {
		cout << "Containers do not match " << endl;
	}
}
```


The Mismatch Algorithm

```
int main() {
	vector<string> vecOne;
	vecOne.push_back("abc");
	vecOne.push_back("def");
	vecOne.push_back("ghi");
	vecOne.push_back("jkl");
	vecOne.push_back("mno");

	vector<string> vecTwo;
	vecTwo.push_back("abc");
	vecTwo.push_back("def");
	vecTwo.push_back("ghi");
	vecTwo.push_back("jkl");
	vecTwo.push_back("mno");
```

```
	vector<string> vecThree;
	vecThree.push_back("abc");
	vecThree.push_back("def");
	vecThree.push_back("ghi");
	vecThree.push_back("zzz");
	vecThree.push_back("jkl");
	vecThree.push_back("mno");

	showMismatchAlgotithm(vecOne,vecTwo);
	showMismatchAlgotithm(vecOne,vecThree);
}
```

The Mismatch Algorithm

```
void showMismatchAlgotithm(vector<string> p1,vector<string> p2) {

    typedef vector<string>::iterator strIterType;
    typedef pair<strIterType,strIterType> resultType;

    resultType result = mismatch(p1.begin(),p1.end(),p2.begin());

    if(result.first == p1.end()) {
        cout << "Containers match!" << endl;
    } else {
        cout << "Containers stop matching at position "
                << result.first - p1.begin() << endl;
    }
}
```


The Search Algorithms

- The search algorithm is very similar to ‘find_end’
  - You provide iterators to the stand and end of a main sequence and a subsequence to be found
  - A predicate can be used to define equality
  - Unlike ‘find_end’ an iterator to the first match is returned
- The ‘search_n’ algortihmn finds equal values
  - It searches the main sequence for a sequence of equal values
  - You specify the number of equal values in the subsequence
  - If not found an iterator to the end of the sequence is returned

The Search Algorithm

```
void showSearchAlgorithm() {
    vector<string> vec;
    vec.push_back("ab");
    vec.push_back("cd");
    vec.push_back("ef");
    vec.push_back("gh");
    vec.push_back("ij");
    vec.push_back("kl");

    vector<string> target;
    target.push_back("ef");
    target.push_back("gh");
    target.push_back("ij");

    vector<string>::iterator result;
    result = search(vec.begin(),vec.end(),target.begin(),target.end());
    if(result == vec.end()) {
        cout << "Sequence cannot be found" << endl;
    } else {
        cout << "Sequence found at " << result - vec.begin() << endl;
    }
}
```

The Search N Algorithm

```
void showSearchNAlgorithm() {
	vector<string> vec;
	vec.push_back("ab");
	vec.push_back("cd");
	vec.push_back("ef");
	vec.push_back("gh");
	vec.push_back("xx");
	vec.push_back("xx");
	vec.push_back("xx");
	vec.push_back("xx");
	vec.push_back("ij");
	vec.push_back("kl");

	vector<string>::iterator result;
	result = search_n(vec.begin(),vec.end(),4,"xx");
	if(result == vec.end()) {
		cout << "Sequence cannot be found" << endl;
	} else {
		cout << "Sequence found at " << result - vec.begin() << endl;
	}
}
```


The Mutating STL Algorithms

Algorithm	Description
copy	Copies an element from one sequence to another
swap	Swaps elements in two containers
transform	Performs an operation on values before copying them
replace	Replaces one value with another
fill	Fills a sequence with a single value
generate	Fills a sequence with the result of an operation
remove	Reorders a sequence to get rid of unwanted values
unique	Reorders a sequence to get rid of duplicated values
reverse	Reverses the order of elements in the sequence
rotate	Rotates values around a midpoint
random_shuffle	Randomizes elements in a sequence
partition	Splits elements in a sequence into two groups

The Copy Algorithm

- This copies values from one sequence into another
  - You supply iterators to mark the start and end of the sequence to be copied and the position in the target to begin copying
  - The 'copy_backward' algorithm inserts the elements to be copied from last to first
- How the copy works depends on the third iterator
  - If it is an output operator elements in the target are overwritten
  - If it is an insertion iterator then new elements are added

The Copy Algorithm

```
int main() {
    list<string> listOne;
    listOne.push_back("abc");
    listOne.push_back("def");
    listOne.push_back("ghi");

    list<string> listTwo;
    listTwo.push_back("zzz");
    listTwo.push_back("yyy");
    listTwo.push_back("xxx");

    back_insert_iterator< list<string> > back_iter(listOne);
    front_insert_iterator< list<string> > front_iter(listOne);

    copy(listTwo.begin(),listTwo.end(),back_iter);
    showContents(listOne,cout);
    copy(listTwo.begin(),listTwo.end(),front_iter);
    showContents(listOne,cout);
}
```


The Swap Algorithm

- The ‘swap’ algorithm exchanges elements
  - Either in the same container or in different containers
- There are three versions
  - ‘swap’ exchanges values via references
  - ‘iter_swap’ exchanges values via iterators
  - ‘swap_ranges’ swaps a sequence of values
    - We provide start and end iterators to the sequence in one container and a third iterator to where the swap should start
    - An iterator is returned which points to the start of the sequence in the second container

The Swap Algorithm

```
int main() {
    list<string> listOne;
    listOne.push_back("abc");
    listOne.push_back("def");
    listOne.push_back("ghi");

    list<string> listTwo;
    listTwo.push_back("zzz");
    listTwo.push_back("yyy");
    listTwo.push_back("xxx");

    swap_ranges(listOne.begin(),listOne.end(),listTwo.begin());

    showContents(listOne,cout);
}
```

Transform and Replace Algorithms

- The 'transform' algorithm is similar to 'copy'
  - Except that you need to apply an operation to each value and put the result in the second container
  - You can use a unary function to transform each value or a binary function to transform elements from two sequences
- The 'replace' algorithm finds and replaces a value
  - The replacement value is just a simple variable
  - The 'replace_if' variant uses a predicate for searching
- Alternate versions of replace insert into a new container
  - The original sequence of values are left untouched
  - These are 'replace_copy' and 'replace_copy_if'

The Transform Algorithm

```
int timesTwo(int value) {
	return value *= 2;
}

int main() {
	vector<int> input;
	input.push_back(10);
	input.push_back(20);
	input.push_back(30);
	input.push_back(40);
	input.push_back(50);

	transform(input.begin(),input.end(),input.begin(),timesTwo);

	showContents(input,cout);
}
```


The Replace Algorithm

```
bool lessThanThirty(int value) {
	return value < 30;
}
int main() {
	vector<int> input;
	input.push_back(10);
	input.push_back(50);
	input.push_back(20);
	input.push_back(30);
	input.push_back(40);
	input.push_back(50);

	replace(input.begin(),input.end(),50,180);
	replace_if(input.begin(),input.end(),lessThanThirty,200);

	showContents(input,cout);
}
```

The Fill and Generate Algorithms

- These are designed for initializing a sequence
  - They are more convenient than using a 'for_each'
- The 'fill' algorithm inserts the same value value into the range between two iterators
  - The 'fill_n' variant fills n positions after an iterator
- The 'generate' algorithm uses a predicate to produce values to add to the specified range
  - Using a functor is more convenient that a simple function because you can cache state inside the fields
  - Again there is a 'generate_n' variant which fills in n positions after the starting iterator

The Fill Algorithm

```
int main() {
	vector<string> vec(12);

	fill(vec.begin(),vec.end(),"aaa");
	showContents(vec,cout);

	cout << endl << endl << "----------------" << endl << endl;

	fill_n(vec.begin(),6,"zzz");
	showContents(vec,cout);
}
```


The Generate Algorithm

```
class Generator {
	string token;
	string state;
public:
	Generator(char *str) : state(str), token(str) {}
	string operator()() {
		return state += token;
	}
};
int main() {
	vector<string> vec(12);

	Generator gen1("a");
	generate(vec.begin(),vec.end(),gen1);
	showContents(vec,cout);

	cout << endl << endl << "----------------" << endl << endl;

	Generator gen2("A");
	generate_n(vec.begin(),6,gen2);
	showContents(vec,cout);
}
```

The Remove & Unique Algorithms

- What these algorithms do is frequently misunderstood
  - 'remove' allows you to remove values from a list whereas 'unique' removes repeated adjacent values
- Nothing is actually removed from the container
  - Deleting an element requires a call to a method and the template cannot know which method to call
- Instead deleted elements are overwritten
  - They return an iterator to where the adjusted sequence ends
  - However the actual length of the container has not changed
- You must manually delete the extra elements
  - E.g. 'vec.erase(remove(vec.begin(),vec.end(),"xx"),vec.end())'

The Remove Algorithm

```
int main() {
	vector<string> vec;
	vec.push_back("ab");
	vec.push_back("xx");
	vec.push_back("cd");
	vec.push_back("ef");
	vec.push_back("xx");
	vec.push_back("xx");
	vec.push_back("gh");
	vec.push_back("ij");
	vec.push_back("xx");
	vec.push_back("kl");

	vector<string>::iterator new_end_iter;
	new_end_iter = remove(vec.begin(),vec.end(),"xx");

	showContents< vector<string> >(vec.begin(),new_end_iter,cout);
	showContents(vec,cout);
}
```


The Unique Algorithm

```
int main() {
	vector<string> vec;
	vec.push_back("ab");
	vec.push_back("ab");
	vec.push_back("cd");
	vec.push_back("cd");
	vec.push_back("ef");
	vec.push_back("ef");
	vec.push_back("gh");
	vec.push_back("gh");

	vector<string>::iterator new_end_iter;
	new_end_iter = unique(vec.begin(),vec.end());

	showContents< vector<string> >(vec.begin(),new_end_iter,cout);
	cout << endl << endl << "----------" << endl << endl;
	showContents(vec,cout);
}
```

The Utility Algorithms

- Four algorithms perform common transformations
  - Reversing, rotating, shuffling and splitting a sequence
- 'reverse' reverses the elements
  - The first is swapped with the last and so on
- 'rotate' shifts values in the sequence
  - Values are moved to the left but added to the end of the sequence when they fall of the end
- 'random_shuffle' reorders the sequence randomly
- 'partition' splits the sequence into two subsequences
  - You provide a predicate that determines how the split works
  - It returns an iterator to the boundary between sequences

The Reverse Algorithm

```
int main() {
	vector<string> vec;
	vec.push_back("ab");
	vec.push_back("cd");
	vec.push_back("ef");
	vec.push_back("gh");
	vec.push_back("ij");

	reverse(vec.begin(),vec.end());
	showContents(vec,cout);

	cout << endl << "---------" << endl;

	reverse(vec.begin(),vec.end());
	showContents(vec,cout);
}
```


The Rotate Algorithm

```
int main() {
	vector<string> vec;
	vec.push_back("ab");
	vec.push_back("cd");
	vec.push_back("ef");
	vec.push_back("gh");
	vec.push_back("ij");
	vec.push_back("kl");

	for(unsigned int i=0;i< vec.size(); i++) {
		vector<string>::iterator begin = vec.begin();
		vector<string>::iterator middle = begin + 1;
		rotate(begin,middle,vec.end());
		showContents(vec,cout);
		cout << endl << "---------" << endl;
	}
}
```

The Random Shuffle Algorithm

```
int main() {
      vector<string> vec;
      vec.push_back("ab");
      vec.push_back("cd");
      vec.push_back("ef");
      vec.push_back("gh");
      vec.push_back("ij");
      vec.push_back("kl");

      for(unsigned int i=0;i< 4; i++) {
            random_shuffle(vec.begin(),vec.end());
            showContents(vec,cout);
            cout << endl << "---------" << endl;
      }
}
```


The Partition Algorithm

```
int main() {
      vector<int> vec;
      vec.push_back(10);
      vec.push_back(90);
      vec.push_back(20);
      vec.push_back(80);
      vec.push_back(70);
      vec.push_back(60);
      vec.push_back(50);
      vec.push_back(40);

      vector<int>::iterator partition_iter;
      partition_iter = partition(vec.begin(),vec.end(),greaterThanFifty);

      showContents<vector<int> >(vec.begin(),partition_iter,cout);
      cout << endl << "--------" << endl;
      showContents<vector<int> >(partition_iter,vec.end(),cout);
}
```


Sorting and Merging Algorithms

Algorithm	Description
sort	Performs a quicksort on a set of values
stable_sort	Performs a sort which preserves the ordering of identical values
partial_sort	Brings n smallest elements to the start of the sequence
nth_element	Finds and positions the n'th largest or smallest element
binary_search	Performs a binary search
lower_bound	Finds the first position where a value can be inserted in order
upper_bound	Finds the last position where a value can be inserted in order
equal_range	Finds the largest subrange where a value can be inserted in order
merge	Merges two sequences in order of increasing value
inplace_merge	Merges sequences in place

Sorting and Merging Algorithms

- The 'sort' algorithm performs a quicksort
  - This is what you should use for most sorting tasks
- Other algorithms are provided for special cases
  - 'stable_sort' keeps identical elements in the same order in which they were originally found
  - 'partial_sort' sorts only a subset of the sequence
  - 'nth_element' moves a single element into its sorted position
- These algorithms cannot be used with lists
  - Because they require random access iterators
  - List provides a 'sort' method which does a stable sort

Sorting a Container

```
int main() {
	vector<string> vecOne;
	loadTestData(vecOne);

	sort(vecOne.begin(),vecOne.end());
	showContents(vecOne,cout);
	cout << endl << "--------" << endl;

	vector<string> vecTwo;
	loadTestData(vecTwo);

	vector<string>::iterator middle = vecTwo.begin() + 5;
	partial_sort(vecTwo.begin(),middle,vecTwo.end());
	showContents(vecTwo,cout);
}
```


Set and Numeric Algorithms

Algorithm	Description
includes	Tests if each member of S2 is contained in S1
set_union	Builds a structure containing the union of S1 and S2
set_intersection	Builds a structure containing shared values in S1 and S2
set_difference	Builds a structure containing values in S1 but not S2
set_symmetric_difference	Builds a structure containing non-shared values of S1 and S2

Algorithm	Description
accumulate	Adds all values in a sequence to an initial value
partial_sum	Adds all the values in a sequence, storing the result of each addition into a separate sequence
adjacent_difference	Finds the difference between values in a sequence, storing each result into a separate sequence
inner_product	Adds the inner product of two containers to an initial value

Introducing C++11

Part 1 – Language Features

garth@ggilmour.com

The New Version of C++

- A new version of C++ has just been released
  - A list of possible features was published in 2005 ('TR1')
  - A draft standard was first published in 2008
  - It was ratified this year as C++11
- There are many important enhancements
  - Which reflect how the language is being used today
- It will be a long time before it is commonly used
  - Most teams will need to support legacy compilers
  - However you can start exploring new features today
    - 'Visual Studio 2010' and 'g++' have support built in
    - Not all features are supported (e.g. delegating constructors)

Enhancing Type Safety

- Many new features prevent type errors:
  - The 'nullptr' keyword denotes a value of type 'nullptr_t'
    - This can be converted to any pointer type, but not to numbers
    - Removing the confusion about using zero as a memory address
  - Enumerations are now strongly typed
    - When you use the 'class' keyword in the declaration
    - E.g. '**enum class** Colors { RED, GREEN BLUE }'
  - The '>>' token is now parsed sensibly
    - Preventing the famous bug when closing nested templates
  - UTF-8, 16 and 32 bit char encodings are all supported
    - String literals would be 'u8"abc"', 'u"abc"' and 'U"abc"'
  - The 'long long int' type has been standardized

Simplifying Object Creation

- Constructors in the same class can call one another
  - This removes the need to create private helper functions
- There is a new unified initialization syntax
  - Both structs and classes share a common syntax
    - In the case of a structure the fields are initialized directly
    - In the case of a class the matching constructor is called
  - So 'Foo {123, "abc", false}' works in both cases
    - The '{ … }' is of type 'std::initializer_list'
- There is a new type of reference
  - This is known as an 'rvalue reference' and denoted by '&&'
  - It enables you to move resources without creating copies


```
struct MyStruct {
    int i;
    double d;
    bool b;

    void print(ostream& sink) {
        sink << i << " " << d << " " << b << endl;
    }
};
class MyClass {
public:
    MyClass(short s, float f, const string & str) {
        this->s = s;
        this->f = f;
        this->str = str;
    }
    void print(ostream& sink) {
        sink << s << " " << f << " " << str << endl;
    }
private:
    short s;
    float f;
    string str;
};
```

```
int main() {
    MyStruct ms {12, 3.4, true};
    MyClass  mc {56, 7.8f, "abc"};

    ms.print(cout);
    mc.print(cout);
}
```


Type Inference

- C++11 introduces type inference
  - When you say '**auto** myvar = func()' the compiler works out the type of 'myvar' for you, thus preserving strong typing
  - This is helpful in several ways:
    - When working with templates with many type parameters
    - When trying meta-programming techniques within C++
- There is also 'decltype' for template authors
  - When you say '**decltype**(foo) bar' the compiler infers 'foo' and declares 'bar' to be of the same type
    - So you are saying "I don't know the type of 'foo' but I do know that 'bar' needs to be of that type as well, so declare it that way"
  - This is particularly useful for the return values of templates

```
class MyClass {
public:
	MyClass(int value) : value(value) {}
	int getValue() { return value; }
private:
	int value;
};
map<string, vector<MyClass>> * func() {
	vector<MyClass> theVector;
	theVector.push_back(MyClass(12));
	theVector.push_back(MyClass(13));
	theVector.push_back(MyClass(14));
	theVector.push_back(MyClass(15));
	theVector.push_back(MyClass(16));

	map<string, vector<MyClass>> * data = new map<string, vector<MyClass>>();
	data->insert(pair<string, vector<MyClass>>("ab12",theVector));
	data->insert(pair<string, vector<MyClass>>("cd34",theVector));
	data->insert(pair<string, vector<MyClass>>("ef56",theVector));

	return data;
}
```



```
int main() {
	auto myvar1 = 12;
	auto myvar2 = new string("abc");
	auto myvar3 = func();

	myvar1 += 3;
	(*myvar2) += "def";
	myvar3->at("cd34").push_back(17);
	myvar3->at("cd34").push_back(18);
	myvar3->at("cd34").push_back(19);

	cout << "Modifed data is:" << endl;
	cout << "\tmyvar1:" << myvar1 << endl;
	cout << "\tmyvar2:" << *myvar2 << endl;

	cout << "\tmyvar3:" << endl;
	for(auto iterator1 = myvar3->begin(); iterator1 != myvar3->end(); iterator1++) {
		auto row = *iterator1;
		cout << "\t\t" << row.first << " holding:" << endl;
		for(auto iterator2 = row.second.begin(); iterator2 != row.second.end(); iterator2++) {
			cout << "\t\t\t" << iterator2->getValue() << endl;
		}
	}
	delete myvar2;
	delete myvar3;
}
```


The For-Each Loop

- C++11 adds a range based for loop
  - It supports any type that has 'begin' and 'end' functions in scope which return iterator objects
    - This includes all the STL container types
  - Naturally arrays are supported as well
- The new loop works well with 'auto'
  - E.g. '**for**(**auto** x : myvec) func(x)'

```
for(int i : myvector) {
	func(i);
}
```

⇨

```
auto start_iter = begin(myvector);
auto end_iter = end(myvector);
while(start_iter != end_iter) {
	int i = *start_iter;
	func(i);
	++start_iter;
}
```

Lambda Functions / Expressions

- The C++ STL encourages you to work with function pointers and/or function objects (aka functors)
  - But this can lead to your code becoming cluttered with lots of short, single-purpose classes and functions
- C++11 addresses this via lambdas
  - The syntax is '[](int a, int b) { return a + b;}'
  - Parenthesis are optional then there are no parameters
    - So '[]() { return func(); }' becomes '[] { return func(); }'
  - A return type is needed when there are multiple lines
    - E.g. '[](int x, int y) -> double { foo(x); bar(y); return zed(); }'
    - This syntax can also be used in normal function pointers


```
void demo1() {
    auto simpleLambda = [](int a, int b) { return (a <= b) ? "foo" : "bar"; };
    cout << simpleLambda(20,30) << endl;
    cout << simpleLambda(30,20) << endl;
}
void demo2() {
    auto multiLineLambda = [](int a, int b)->const char * {
                                if(a <= b) {
                                    return "foo";
                                } else {
                                    return "bar";
                                }
                            };
    cout << "------" << endl;
    cout << multiLineLambda(20,30) << endl;
    cout << multiLineLambda(30,20) << endl;
}
```

```
foo
bar
------
foo
bar
------
foobar
```



```
void demo3() {
    auto noArgsLambda = [] { return "foobar"; };
    cout << "------" << endl;
    cout << noArgsLambda() << endl;
}
void demo4() {
    int a = 12;
    auto lambdaCapturingByRef = [&]()->int {
                                    a = a * 2;
                                    return a;
                                };
    cout << "------" << endl;
    cout << lambdaCapturingByRef() << endl;
    cout << "Local variable now: " << a << endl;
}
```

```
foobar
------
24
Local variable now: 24
```

```
void demo5() {
    int b = 34;
    auto lambdaCapturingByValue = [=]() mutable ->int {
                                      b = b * 2;
                                      return b;
                                  };
    cout << "------" << endl;
    cout << lambdaCapturingByValue() << endl;
    cout << "Local variable now: " << b << endl;
}
void demo6() {
    int c = 56;
    int d = 78;
    auto lambdaWithExplicitCapturing = [c, &d]()->int {
                                           d++;
                                           return c + d;
                                       };
    cout << "------" << endl;
    cout << lambdaWithExplicitCapturing() << endl;
    cout << "Local variables now: " << c << " and " << d << endl;
}
```

```
68
Local variable now: 34
------
135
Local variables now: 56 and 79
```


Improvements to Templates

- You can prevent a template being multiply instantiated
  - If you write 'vector<MyClass> mc;' in several C++ files then an instance is created in each Translation Unit
    - This increases compilation time and can cause errors when combining templates with static fields and dynamic libraries
  - Writing 'extern template class vector<MyClass>;' prevents an instance being created within the current TU
- You can use typedefs without all the type parameters
  - For 'template <typename T, typename U> class X' you can say 'template <typename U> typedef X<string, U> *alias*'

Introducing C++11

Part 2 – The Threading API

garth@ggilmour.com

Threading in C++11

- Threads are not mentioned in the 1998 C++
  - Ironically for a language this is frequently promoted for writing low-level, high performance applications
  - You are dependent on 'pthreads' etc…
  - Thread safety in the STL is library specific
- This is completely changed in C++11
  - A thread-aware memory model is part of the language
  - There are built in methods for launching threads
    - Return values can be acquired via 3 types of future
  - The standard library defines different types of lock
  - You can query thread support on the hardware

Starting Threads

- You launch a thread by creating an 'std::thread' object
  - The constructor takes a 'callable object'
    - This will usually be a function pointer or a functor
      - By functor we mean an object that overloads 'operator()'
    - You can also make use of the new lambda construct
  - The thread is automatically launched for you
    - There is no way to create it in an inactive state
- There are several ways to pass in arguments
  - The std::thread constructor can take additional arguments which match the signature of the 'callable object'
  - Functors can have their state initialized already
  - Lambdas can capture the state around them

Creating threads via functions and functors

```
void myfunc() {
	for(int i=0;i<20;i++) {
		printf("myfunc message %d\n", i);
		this_thread::sleep_for(chrono::seconds(1));
	}
}
class MyFunctor {
public:
	void operator()() {
		for(int i=0;i<20;i++) {
			printf("myfunctor message %d\n", i);
			this_thread::sleep_for(chrono::seconds(1));
		}
	}
};
int main() {
	MyFunctor obj;

	thread t1(myfunc);
	thread t2(obj);

	t1.join();
	t2.join();
	cout << "Threads completed - end of main..." << endl;
}
```

Passing arguments to threads Pt1

```
void myfunc(const char * msg, int count) {
    for(int i=0;i<count;i++) {
        printf("%s message %d\n", msg, i);
        this_thread::sleep_for(chrono::seconds(1));
    }
}
class StatefulFunctor {
public:
    StatefulFunctor(const char * msg, int count) : msg(msg), count(count) {}
    StatefulFunctor(const StatefulFunctor& other) {
        msg = other.msg;
        count = other.count;
    }
    void operator()() {
        for(int i=0;i<count;i++) {
            printf("%s message %d\n", msg, i);
            this_thread::sleep_for(chrono::seconds(1));
        }
    }
private:
    const char * msg;
    int count;
};
```


Passing arguments to threads Pt2

```
class StatelessFunctor {
public:
    void operator()(const char * msg, int count) {
        for(int i=0;i<count;i++) {
            printf("%s message %d\n", msg, i);
            this_thread::sleep_for(chrono::seconds(1));
        }
    }
};
int main() {
    StatelessFunctor obj1;
    StatefulFunctor obj2("Stateful functor",20);

    thread t1(myfunc,"Function",20);
    thread t2(obj1, "Stateless functor", 20);
    thread t3(obj2);

    t1.join();
    t2.join();
    t3.join();
    cout << "Threads completed - end of main..." << endl;
}
```


Running Functions Asynchronously

- Another way to launch a thread is via ‘async’
  - This takes a value from ‘std::launch’ and a function
  - If the value is ‘async’ the function runs on a new thread
    - The other possible values are ‘sync’ and ‘any’
- The value returned from ‘async’ is a future object
  - Also known as an IOU or asynchronous completion token
  - This will hold the true return value when the thread exits
- The true return value can be obtained via ‘future.get’
  - If the value was ‘sync’ this is when the function is called
  - If the value was ‘async’ then the calling thread may block
    - There are multiple ways to test if the future is ‘cooked’ or not

```
struct MyReturn {
    double retval;
    time_t timeCompleted;

    void print(ostream & out) {
        tm * when  = localtime(&timeCompleted);
        out << "\t" << retval << " at "
            << when->tm_hour << ":" << when->tm_min << ":" << when->tm_sec
            << endl;
    }
};

MyReturn threadFunc(int secondsToSleep, double valueToReturn) {
    this_thread::sleep_for(chrono::seconds(secondsToSleep));
    MyReturn r {valueToReturn, time(0) };
    return r;
}
```



```
int main() {
    future<MyReturn> f1 = async(launch::async, []{ return threadFunc(8, 1.23); });
    future<MyReturn> f2 = async(launch::async, []{ return threadFunc(16, 4.56); });
    future<MyReturn> f3 = async(launch::async, []{ return threadFunc(24, 7.89); });

    while(!f1.is_ready() && !f2.is_ready() || !f3.is_ready()) {
        cout << "One or more threads running so main spinning..." << endl;
        this_thread::sleep_for(chrono::seconds(2));
    }

    MyReturn r1 = f1.get();
    MyReturn r2 = f2.get();
    MyReturn r3 = f3.get();

    cout << "Retrieved the results: " << endl;
    r1.print(cout);
    r2.print(cout);
    r3.print(cout);

    return 0;
}
```


Safeguarding Access To Data

- Threads can share data by locking on a mutex
  - Holding multiple mutexes opens the door to deadlocks
- Four types of mutex are available
  - Support is available for a timeout interval and/or re-acquisition of the lock by the currently owning thread
  - The classes are ‘std::mutex’, ‘std::timed_mutex’, ‘std::recursive_mutex’ and ‘std::recursive_timed_mutex’
- Mutexes should be used via a ‘std::lock_guard’
  - This uses the ‘Resource Acquisition is Initialization’ idiom to ensure the mutex is released in all circumstances
  - If you may need to unlock early use an ‘std::unique_lock’

```
void printMsgs(mutex* theMutex, const char * msg, int num) {
        lock_guard<mutex>guard(*theMutex);
        printf("%s iteration %d message A\n", msg, num);
        this_thread::sleep_for(chrono::milliseconds(100));
        printf("%s iteration %d message B\n", msg, num);
        this_thread::sleep_for(chrono::milliseconds(100));
        printf("%s iteration %d message C\n", msg, num);
        this_thread::sleep_for(chrono::milliseconds(100));
}
void myfunc(mutex* theMutex, const char * msg, int count) {
        for(int i=0;i<count;i++) {
                printMsgs(theMutex, msg, i);
                this_thread::sleep_for(chrono::seconds(1));
        }
}
int main() {
        mutex myMutex;
        thread t1(myfunc, &myMutex, "T1", 20);
        thread t2(myfunc, &myMutex, "T2", 20);
        thread t3(myfunc, &myMutex, "T3", 20);

        t1.join();
        t2.join();
        t3.join();
        printf("End of main...");
}
```


Communicating Between Threads

- Threads can communicate via conditional variables
  - Allowing you to implement a producer / consumer design
  - Typically one or more consumer threads wait until notified by another (the producer) that data is available to be read
- Consumer threads wait on the condition
  - The methods are ‘wait’, ‘wait_for’ and ‘wait_until’
  - All of these take a ‘unique_lock’ and optionally a predicate
  - The ‘wait_for’ and ‘wait_until’ version also take time values
- Producer threads call ‘notify_one’ or ‘notify_all’
  - The latter wakes up all threads blocked on the condition


```
void producer(condition_variable * theCondition,
                mutex * theMutex,
                queue<string> * theQueue) {

    string data[] = {"abc","def","ghi","jkl","mno","pqr","xxx"};

    for(int i=0;i<7;i++) {
        theMutex->lock();
        theQueue->push(data[i]);
        printf("Producer just published (will notify in 2 secs)\n");

        this_thread::sleep_for(chrono::seconds(2));

        theCondition->notify_one();
        theMutex->unlock();

        this_thread::sleep_for(chrono::seconds(4));
    }
}
```

```
void consumer(condition_variable * theCondition,
                mutex * theMutex,
                queue<string> * theQueue) {
    while(true) {
        unique_lock<mutex> uniqueLock(*theMutex);

        theCondition->wait(uniqueLock,[theQueue]{ return !theQueue->empty(); });

        string str = theQueue->front();
        theQueue->pop();

        printf("Consumer just read %s\n", str.c_str());
        theMutex->unlock();

        if(str == "xxx") {
            break;
        }
    }
}
```



```
int main() {
    condition_variable myCondition;
    mutex myMutex;
    queue<string> myQueue;

    thread t1(producer, &myCondition, &myMutex, &myQueue);
    thread t2(consumer, &myCondition, &myMutex, &myQueue);

    t1.join();
    t2.join();

    printf("End of main...");
}
```


Other Features of C++ Threads

- Threads can provide a ‘std::thread::native_handle’
  - This allows you to access platform specific functionality
- You can call operations on the current thread
  - E.g. ‘this_thread::sleep_for(TIME)’ and ‘this_thread::yield()’
- Threads can be run in the background as daemons
  - This happens when the associated thread object is destroyed or when you manually call the ‘detach’ method
- Atomic version of the standard types are provided
  - Such as ‘atomic_char’, ‘atomic_int’ and ‘atomic_long’
  - These can have their values changed across multiple threads safely – often without any locking being required

Appendix One

TDD in C++

garth@ggilmour.com

Introducing TDD

- Software is tested in many ways
  - Functional tests make sure the requirements are met
  - Non functional tests make sure each function is Usable, Reliable, has good Performance and is Supportable
  - Acceptance tests represent the criteria set by customers
  - Load testing overwhelms the code with numerous concurrent requests, large volumes of data transfer and slow networks
  - Regression testing validates all the functionality added so far
- Unit tests are the most fundamental kind of test
  - Developers test their own code before they add it to the build

Undisciplined Unit Testing

- Unit testing is rarely systematic
  - Developers do just enough to satisfy themselves
  - This may be a little or a lot…
- Unit tests are mostly junked after use
  - Developers write small test applications (console or GUI based)
    - These serve as ad hoc testing frameworks
  - When these apps have served their purpose they are discarded
    - Because they are ad hoc code they have little reuse value
- This kind of unit testing is never regressive
  - The tests only check the new functionality
  - They fail to ensure that existing functionality is still OK

The Potential of Unit Testing

- Undisciplined unit testing ignores a potential source of massive improvements in code quality
  - The information lost by junking unit tests is priceless
- Systematic unit testing requires two things
  - We retain our unit tests and treat them as artefacts
  - There is a standard pattern for writing tests and a developer friendly set of tools for creating and running them
- Unit tests could then be integrated into our process
  - They can be combined into a test suite which will continuously verify the quality of our code by testing each class in isolation

Test Driven Development

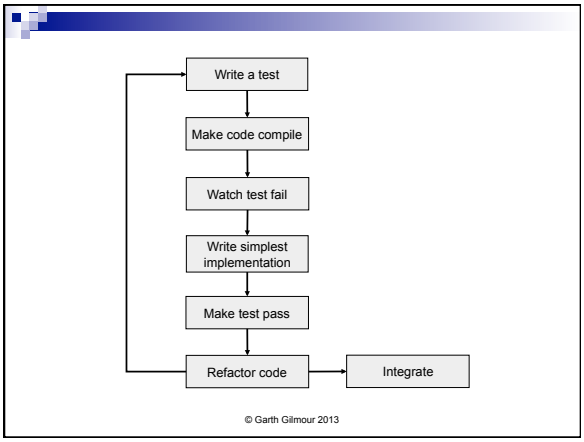
- TDD is disciplined unit testing
  - It provides a methodology, a toolset and a philosophy
- It adds an extra dimension to unit testing
  - The idea that the tests should be written before the code
  - We write tests to define functionality as much as to validate it
- Writing the tests before the code makes good sense
  - It forces you to write code from the clients perspective
    - Leading to simpler and more intuitive interfaces
  - It encourages the developer to work incrementally
    - Preventing hours of 'heroic' programming between tests
  - It encourages you to think about the essentials
    - How to get this test to pass without breaking any of the others

Test Driven Development

- Communicating via test cases also has benefits
  - When new requirements arrive write tests for them
  - When bugs are reported write tests and make them work
  - Use the test cases to learn how to use a particular class
  - Run the suite of tests to make changes with confidence
- Test cases give new developers a starting point
  - Arbitrarily browsing around a new system is inefficient
    - But this is how new developers typically learn a system
  - UML diagrams can be used to discover the major classes, which can then be dissected via the test cases

The TDD Development Cycle

1. Write a test for the new requirement
   - Each test should be as fine grained as possible
   - Don't be afraid to make the tests very simple
2. Make the test work
   - Write just enough code to pass the test
   - This usually leads to a 'but it still wont work for…'
3. Write further tests until the code works right
   - Until you can no longer think of useful tests to add
4. Refactor the code to keep the design simple
   - We will discuss refactoring later
5. Integrate your work with the main build



The TDD Development Cycle

- Three critical points to note are:
  - Celebrate the 'but that's stupid' moments!
    - This is what keeps us going the right way
  - Always watch the test fail first
    - Sometimes your test will run without adding code…
  - Don't implement and refactor at the same time
    - They are two separate activities
    - Mixing them only causes confusion
    - Remember tests need refactoring as well

The TDD Development Cycle

- We do a 'check-in' every time we add functionality
  - Not when we have implemented a complete requirement
- Up front design is kept to a minimum (NOT omitted)
  - We always do the simplest thing that could possibly work
- The system is complete when it passes all the tests
  - If functionality is absent or incomplete write more tests…
- Test cases also need to be maintained
  - To avoid duplicated code and functionality
  - To introduce common ways of loading test data

Keeping a Notebook

- Many possible tests will occur to you as you code
  - Its important not to get distracted by tangential issues
    - Stick with the simplest possible solution
  - But you should always have a notebook handy
    - To jot down ideas for tests, enhancements etc…
- Sometimes multiple tests help scope out the next step
  - This is known as 'triangulation'
- If you are doing TDD right it can become boring
  - Sticking to the simplest solution instead of exploring alternatives
  - Some projects like to leave an 'escape route' for developers
    - A period of time set aside specifically for exploring radical ideas

Test First Verses Test After

- A common question asked about TDD is:
  - "Does it matter if I write my tests after my code?"
- Remember there are three benefits to TDD
  - The interface of each class is kept simple
  - All functionality within the class has tests
  - Other developers can do regression testing
    - This means they can refactor and enhance the code without fear
- If you practice 'Test After Development'
  - Simplicity of design depends on your own intuitions
  - Unless you are very conscientious the tests will have gaps
  - The ability to run regression tests will be unchanged

Limitations of TDD

- TDD does not replace the QA department
  - It does not guarantee that components, layers, subsystems etc… will operate correctly when they are assembled
- However it does simplify the work of the QA team
  - The QA team will not waste time discovering low level bugs
  - Developers should be more willing to involve themselves in QA
  - TDD provides a foundation on which it is possible to build a rigorous set of acceptance tests using feature rich testing tools
- TDD is hard work to maintain under pressure
  - In any software process code quality is the most tempting thing to sacrifice in a crisis and the hardest thing to regain afterwards

Introducing GoogleTest

- It is very difficult to do TDD in C++
  - Or in any language that is not VM based
  - The physical design of the language acts against the concept of working in a tight 'write-test-refactor' loop
- Unit testing tools cannot leverage reflection
  - Hence they have to use some combination of macros, template based meta-programming and RTTI
- There have been my attempts at an 'xUnit' for C++
  - Such as the one provided with the Boost libraries
  - 'GoogleTest' seems to be the best at present

Installing and Running 'GoogleTest'

- The first step is to build the framework from scratch
  - Build files are provided for popular IDE's
  - On Ubuntu you can simply use 'apt-get'
- Next you write a program containing some tests
  - We will see how to do this in just a moment
  - The 'main' method must call 'RUN_ALL_TESTS'
- Finally you build and run your program
  - Making sure the library you built is linked in

```
sudo apt-get install libgtest0 libgtest-dev
```

```
g++ -lgtest_main -lpthread -o demo1 main.cpp
```

```
#include "gtest/gtest.h"

int add(int no1, int no2) {
    return no1 + no2;
}
int subtract(int no1, int no2) {
    return no1 - no2;
}
int multiply(int no1, int no2) {
    return no1 * no2;
}

TEST (MathTests, Addition) {
    ASSERT_TRUE (12 == add(9,3));
    ASSERT_TRUE (23 == add(19,4));
    ASSERT_TRUE (34 == add(31,3));
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```


Writing Simple Unit Tests

- Simple tests use the 'TEST' macro
  - This generates a function that returns 'void'
- The macro takes two inputs:
  - The name of the current test case
  - The name of the test itself
- The two names must be unique when combined
  - Different test cases can have tests with the same name
- When 'RUN_ALL_TESTS' is called:
  - Every test compiled and linked into the program is run
  - Zero is returned if all tests passed and one otherwise

Useful Google Test Options

- Test output can be written to an XML file
  - Via the '--gtest_output="xml:foo.xml"' option
- Subsets of all tests in the program can be run
  - The '--gtest_filter' option takes a colon separated list of wildcard based patterns, representing tests to be run
    - E.g. '--gtest_filter=MathsTests.*:AcceptanceTests.*'
  - There can also be another list preceded with '-' which specifies patterns of tests not to be run
- It is also possible to repeat and shuffle tests
  - Specifying '--gtest_repeat=100' runs each test 100 times
  - Specifying '--gtest_shuffle' runs the tests in a random order

Writing Simple Unit Tests

- As in any 'xUnit' tool your tests contain assertions
  - These check the outputs and state of the code under test
  - In Google Test assertions are written as macros
- Each assertion comes in two forms
  - Macros beginning with 'ASSERT_' abort the function
  - Those beginning with 'EXPECT_' keep going
- The 'EXPECT_' macros are to be preferred
  - Because they enable more than one error to be reported and prevent clean up code being skipped (leading to leeks)
  - Note this is the reverse of the advice given in Java and C#


```
TEST (MathTests, Addition) {
   ASSERT_TRUE (12 == add(9,3));
   ASSERT_TRUE (23 == add(19,4));
   ASSERT_TRUE (34 == add(31,3));
}

TEST (MathTests, Subtraction) {
   ASSERT_EQ (24, subtract(29,5)) << "Subtraction failed for 29 and 5";
   ASSERT_EQ (35, subtract(42,7)) << "Subtraction failed for 42 and 7";
   ASSERT_EQ (68, subtract(77,9)) << "Subtraction failed for 77 and 9";
}

TEST (MathTests, Multiplication) {
   EXPECT_EQ (45, multiply(9,5))   << "Multiplication failed for 9 and 5";
   EXPECT_EQ (168, multiply(56,3)) << "Multiplication failed for 56 and 3";
   EXPECT_EQ (63, multiply(9,7))   << "Multiplication failed for 9 and 7";
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```


Making Use of Assertions

- Custom messages can be provided for assertions
  - By using the '<<' operator to stream text into the macro
- Many different types of assertion are provided
  - In each case the expected value is placed second
  - Switching the order will result in confusing messages
- Take special can when working with strings
  - 'EXPECT_EQ' compares the values given, so when used with 'C' strings it compares the addresses of the arrays
    - When used with 'std::string' objects it behaves as expected
  - Use special functions like 'EXPECT_STREQ' instead

Making Use of Assertions

Macro	Description
ASSERT_TRUE / EXPECT_TRUE ASSERT_FALSE / EXPECT_FALSE	Test a single condition
ASSERT_EQ / EXPECT_EQ ASSERT_NE / EXPECT_NE	Performs a value comparison on two items
ASSERT_LT / EXPECT_LT ASSERT_LE/ EXPECT_LE ASSERT_GT / EXPECT_GT ASSERT_GE / EXPECT_GE	Perform < , <=, > and >= comparisons on two items
ASSERT_STREQ / EXPECT_STREQ ASSERT_STRNE / EXPECT_STRNE	Compares the contents of two char arrays (aka C strings)
ASSERT_STRCASEEQ / EXPECT_STRCASEEQ ASSERT_STRCASENE / EXPECT_STRCASENE	As above but case insensitive

Writing More Complex Tests

- When writing tests we typically want to:
  - Set up test data before the test is run
  - Remove the test data after the test
  - Use helper functions across tests
- This can be achieved by writing a 'Test Fixture'
  - This is class which inherits from '::testing::Test'
  - It contains virtual 'SetUp' and 'TearDown' methods
    - When you override these your implementations should be protected or public so the framework can make use of them
- These tests are run using the 'TEST_F' macro
  - A new instance of the fixture is created for each test


```
struct TestDataType {
    int first;
    int second;
    int third;
};

map<string,TestDataType> testData;

typedef pair<string,TestDataType> Entry;
```

```
class MyTestBase : public ::testing::Test {
protected:

    void SetUp() {
        TestDataType t1 =  {19,12,7};
        TestDataType t2 = {15,19,4};
        TestDataType t3 = {54,6,9};

        testData.insert(Entry("add",t1));
        testData.insert(Entry("subtract",t2));
        testData.insert(Entry("multiply",t3));
    }

    void TearDown() {
        testData.clear();
    }
};
```

```
TEST_F (MyTestBase, Addition) {
    TestDataType values = testData["add"];
    int retval = add(values.second, values.third);
    ASSERT_EQ(values.first,retval) << "Addition failed for specified values";
}
TEST_F (MyTestBase, Subtraction) {
    TestDataType values = testData["subtract"];
    int retval = subtract(values.second, values.third);
    ASSERT_EQ(values.first,retval) << "Subtraction failed for specified values";
}
TEST_F (MyTestBase, Multiplication) {
    TestDataType values = testData["multiply"];
    int retval = multiply(values.second, values.third);
    ASSERT_EQ(values.first,retval) << "Multiplication failed for specified values";
}
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```


Additional Test Fixture Functionality

- A test fixture instance is created for each test
  - In order to ensure that tests are independent of each other
  - This means that resources are continually being created
- You can avoid this via static methods
  - If you declare static 'SetUpTestCase' and 'TearDownTestCase' methods they will be called once
    - Before the first test or after the last test in the test case
- There is an easy way to disable problematic tests
  - If you place 'DISABLED' in front of the class name then none of the 'TEST_F' tests based on that class will run
  - A warning is output so you know tests have been skipped

Going Beyond Test Fixtures

- If multiple test cases share resources you can:
  - Create a class which extends '::testing::Environment'
  - Override its 'SetUp' and 'TearDown'methods
  - Register your class via 'AddGlobalTestEnvironment'
- It is possible to listen to tests as they are run
  - Derive from 'TestEventListener' and override the callbacks
    - This is a class holding only pure virtual methods (an interface)
    - There is also the 'EmptyTestEventListener' convenience class
  - Register an instance of your listener in 'main'
    - '::testing::UnitTest::GetInstance()->listeners().Append(XXXX)'

Additional Types of Assertion

- There are three macros for signaling failure
  - Used in branches of conditionals that should not execute
  - 'FAIL' causes the test function to be aborted
  - 'ADD_FAILURE' and 'ADD_FAILURE_AT' generate a failure but don't abort the test function
- There are also macros for exception handing:
  - 'ASSERT_THROW' checks for a specified exception type
  - 'ASSERT_ANY_THROW' checks for any type of exception
  - 'ASSERT_NO_THROW' checks that nothing is thrown

Additional Types of Assertion

- There are six assertions for floating point comparisons
  - ASSERT_FLOAT_EQ, ASSERT_DOUBLE_EQ and their 'EXPECT_*' versions check that two values are 'almost equal'
    - That is they are within 4 'Units in the Last Place'
  - ASSERT_NEAR lets you specify an error boundary manually
- It is possible to write 'death tests'
  - Tests which verify that your application exits with an error message when some erroneous code executes
  - Assertions such as 'ASSERT_DEATH' cause the framework to spawn a new process and run the specified code
    - You can check the error message and/or return value of the process

Extra Google Test Functionality

- Assertions can be enhanced in various ways:
  - You can ask it to run your own functions as assertions
  - Functions in asserts can return 'AssertionResult' objects
  - Predicate formatter functions can be used to fully customize the format of error messages
- There are also advanced types of test:
  - A 'value parameterized' test is where the same function is called repeatedly for a range of values
  - A 'typed test' is where the same function is called repeatedly for a range of different data types
    - This is useful when writing your own generic functions / classes

Why Unit Testing Requires Mocks

- Hopefully unit testing sounds cool…
  - But if you try to apply what you have learnt so far to a typical commercial codebase you will run into a big problem
- All the classes we have looked at are stand-alone
  - Real world classes have dependencies
  - How do you unit test class 'A' if it cannot do its job without instances of 'B', 'C' and 'D' ?
- Fortunately there are ways around this problem
  - Open source extensions to unit testing frameworks
  - Building and inserting Mock Objects

Introducing Mock Objects

- Mock Objects is the standard solution to the problem of unit testing classes with dependencies
  - A Mock Object is a class which implements the same interface as the real dependency, but whose methods return test data
- The unit test object 'injects' the object to be tested with mock objects rather than 'real' ones
  - If the dependencies are of interface types then the object being tested has no way of distinguishing between them
  - The unit test knows what data the mock objects will return and can ask them when and how they were called

Introducing Mock Objects

- Mocking uses the concept of 'Dependency Injection'
  - Your classes should not create their own dependencies but instead accept them from higher levels in the design
- Fortunately DI is an OO best practice
  - If 'A' uses 'B' and 'B' is dependent on 'C' then 'A' should present the 'C' to 'B' and the coupling should be light
  - I.e. the 'B -> C' dependency should be expressed through an interface (or pure virtual base class as it is in C++)
- Open source tools now exist to build mock objects
  - Which can be 'programmed' with the desired behaviors

A Key Principle in Mocking

- Most people think about mocks the wrong way
  - They assume that the mock will be 'stupid'
  - I.e. it will only return hard coded values or log to a file
- There are many kinds of mock objects…
  - Don't think of them as just returning hard-coded values
  - Think instead of a range of mock objects for different purposes
- The most useful mock objects can be given:
  - Expectations about how they will be used by the client
  - Instructions about what to do when they are called

The Different Kinds of Mock Object

- The term 'Test Double' has been coined
  - To describe all the different kinds of mock object
- 'Mock' is reserved for intelligent mock objects
  - Which can be programmed with expectations
- The opposite of a 'Mock' is a 'Stub'
  - This only returns hard-coded values
- A 'Crash Test Dummy' is a 'Stub' that raises errors
  - Every method call raises an exception, returns null etc…
  - This allows the unit test to verify exception handling behaviour

The Different Kinds of Mock Object

- A 'Fake' is a simplified implementation
  - It allows the class under test to:
    - Use a real implementation of an API
    - Work with sample data
  - However all the resources are local to the current machine
    - So startup/shutdown times are not as significant
- Examples of fakes (in Java) would be:
  - JDBC, as provided by an in-memory database (Java DB)
    - This can also be used for Hibernate, JPA etc…
  - HTTP, as provided by a lightweight server (Tomcat)
    - This can also be used for JAX-WS based Web Services

Introducing GoogleMock

- Google Mock is a C++ specific mocking framework
  - Inspired by the 'jMock' and 'EasyMock' frameworks
  - It uses macros to generate mock objects from interfaces
- The objects generated are mocks in the TDD sense
  - When you have created a mock you can program it to have expectations about how it will be used
  - Once again this is accomplished via macros
- Google Mock was designed for use with Google Test
  - But it can now be combined with other testing frameworks

```
sudo apt-get install google-mock
```



```
class Shop {
public:
   Shop(StockCheckEngine * stockEngine) {
      this->stockEngine = stockEngine;
   }
   bool purchaseItems(map<string,int> theItems, vector<string>& itemsNotAvailable) {
      bool retval = true;

      map<string,int>::iterator iter;
      for(iter = theItems.begin(); iter != theItems.end(); iter++) {
         string itemId = iter->first;
         int quantityRequired = iter->second;
         int quantityInStock = stockEngine->checkQuantityInStock(itemId);

         cout << "Require " << quantityRequired << " of " << itemId  << " found " << quantityInStock << endl;
         if(quantityInStock < quantityRequired) {
            itemsNotAvailable.push_back(itemId);
            retval = false;
         }
      }
      return retval;
   }
private:
   StockCheckEngine * stockEngine;
};
```

This is the code to be tested (note dependency)

Creating a Mock Object

- The best place to start is with a pure virtual base class
  - There is a template based approach for mocking concrete methods, but it is much more complex
- We derive a new class from the pure virtual base
  - This will contain a macro for each method to be mocked
- When mocking methods we:
  - Declare a macro of type 'MOCK_METHODn'
    - Where 'n' is the number of arguments the method takes
  - Pass the name as the first argument to the macro
  - Pass the signature as the second argument to the macro

```
class StockCheckEngine {
public:
    virtual int checkQuantityInStock(string id) = 0;
};

class MockStockCheckEngine : public StockCheckEngine {
public:
    MOCK_METHOD1( checkQuantityInStock, int(string id) );
};
```

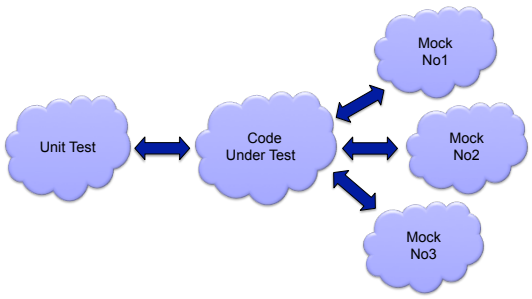
Macro: note the number at the end of the name

Method Name

Method Signature

Using the Mock Objects

- Inside our test class we:
  - Create instances of the mock objects
  - Configure their default behavior via 'ON_CALL'
  - Define expectations via 'EXPECT_CALL'
- Both these macros use a fluent interface
  - We chain methods together to create a sentence-like effect
  - E.g. 'EXPECT_CALL(…).After(…).With(AllOf(…)).'
- It helps to think of the code under test as surrounded:
  - The unit test class is on one side testing patterns of usage
  - Mock objects are on the other simulating dependencies



```
TEST (ShopTests, PurchaseWithAllItemsInStock) {
   MockStockCheckEngine mse;
   Shop shop(&mse);
   EXPECT_CALL(mse, checkQuantityInStock("AB12"))
                        .Times(Exactly(1)).WillOnce(Return(20));
   EXPECT_CALL(mse, checkQuantityInStock("CD34"))
                        .Times(Exactly(1)).WillOnce(Return(21));
   EXPECT_CALL(mse, checkQuantityInStock("EF56"))
                        .Times(Exactly(1)).WillOnce(Return(22));
   map<string,int> itemsToOrder;
   itemsToOrder.insert(pair<string,int>("AB12",15));
   itemsToOrder.insert(pair<string,int>("CD34",16));
   itemsToOrder.insert(pair<string,int>("EF56",17));

   vector<string> itemsUnderstocked;
   EXPECT_TRUE(shop.purchaseItems(itemsToOrder,itemsUnderstocked));
   EXPECT_EQ(0,itemsUnderstocked.size()) << "No items should be understocked";
}
```

Programming mock objects via the fluent interface


```
TEST (ShopTests, PurchaseWithSomeUnderstockedItems) {
   MockStockCheckEngine mse;
   Shop shop(&mse);

   EXPECT_CALL(mse, checkQuantityInStock("AB12"))
                        .Times(Exactly(1)).WillOnce(Return(14));
   EXPECT_CALL(mse, checkQuantityInStock("CD34"))
                        .Times(Exactly(1)).WillOnce(Return(21));
   EXPECT_CALL(mse, checkQuantityInStock("EF56"))
                        .Times(Exactly(1)).WillOnce(Return(16));

   map<string,int> itemsToOrder;
   itemsToOrder.insert(pair<string,int>("AB12",15));
   itemsToOrder.insert(pair<string,int>("CD34",16));
   itemsToOrder.insert(pair<string,int>("EF56",17));

   vector<string> itemsUnderstocked;
   EXPECT_FALSE(shop.purchaseItems(itemsToOrder,itemsUnderstocked));
   EXPECT_EQ(2,itemsUnderstocked.size()) << "Two items should be understocked";
   EXPECT_EQ(string("AB12"),itemsUnderstocked[0]);
   EXPECT_EQ(string("EF56"),itemsUnderstocked[1]);
}
```

Checking the alternate path

The Fluent Interface in Depth

- To program a mock object we must tell it:
  - What arguments to expect when called
  - How many times if should expect to be called
  - What actions to perform each time it is called
- The basic syntax is shown below
  - A lot of variation is possible to enable powerful mocks
  - As noted previously the calls form a 'fluent interface'
  - Also known as an 'internal domain specific language'

```
EXPECT_CALL(mockName, methodName(matchers))
                        .Times(cardinality)
                        .WillOnce(action)
                        .WillRepeatedly(action);
```

Specifying the Arguments

- Arguments can be specified as exact values
  - But this can lead to bloated configuration and is unsuitable for arguments that accept a variety of valid values
- A better solution is to use matchers
  - These are functions that perform a check on the argument
    - Additional functions exist to let you combine matchers
    - See the tables on the following slide for details
  - Additionally the symbol '_' acts as the wildcard
- Matchers usually apply to a single argument
  - But you can insert a call to 'With' before 'Times'
  - 'With' accepts matchers that compare arguments

Matcher Function	Description
Eq(x), Ne(x)	Tests if the argument is equal or not equal to 'x'
Le(x), Lt(x), Ge(x), Gt(x)	Compares the value of the argument to 'x'
FloatEq(x), DoubleEq(x)	Special equality tests for floating point values
IsNull, NotNull	Tests if argument is the null pointer
StrEq(x), StrNe(x)	Special equality tests for strings (C or C++)
StrCaseEq(x), StrCaseNe(x)	Case insensitive versions of the above
ContainsRegex(x), MatchesRegex(x), StartsWith(x), EndsWith(x)	Special comparisons for strings
Contains(x), Each(x)	Tests items in a container ('x' can be a matcher)
ElementsAre(…), ElementsAreArray(x)	Tests the whole content of a container
ResultOf(f, m)	'f' is called with the current argument and the value returned is checked against 'm'
AllOf(…), AnyOf(…)	Allow you to combine multiple matchers

Specifying a Cardinality

- The call to 'Times' specifies the cardinality
  - 'Times(0)' disallows a call with the given arguments
- Often the cardinality will be 'fuzzy'
  - In which case you can use one of the functions below
- Cardinality will be inferred if 'Times' is not called
  - E.g. if you call 'WillOnce' 3 times then 3 is the cardinality

Cardinality Method	Description
AnyNumber	Any number of calls are allowed
AtLeast(n), AtMost(n)	Places a lower or upper bound on the number of calls
Between(n,m)	Specified a range for the number of calls
Exactly(n)	Specifies exactly the number of calls

Specifying Actions

- Actions are set via 'WillOnce' and 'WillRepeatedly'
  - 'WillOnce' can be called any number of times
  - 'WillRepeatedly' can be called at most once
- Actions are specified as functions
  - It is possible to perform multiple actions via 'DoAll'
  - The action is evaluated once, not on each invocation
    - E.g. 'WillRepeatedly(Return(x++))' always gives the same value
- Using actions you can:
  - Return a hardcoded value or a heap allocated object
  - Invoke one of the arguments or an arbitrary function
  - Delete one of the arguments or modify its value

Action Function	Description
Return()	Return from a void function
Return(x), ReturnNull()	Return x or the null pointer
ReturnArg<N>()	Return the value of the N-th argument
ReturnNew<T>(…)	Allocate a new T with the specified constructor arguments, a new object is created on each call
Assign(&variable, value)	Assign a value to an arbitrary variable
SetArgReferee<N>(value) SetArgPointee<N>(value)	Assign a value to a variable passed by reference (i.e. the argument is a C++ pointer or reference)
Throw(x)	Throw 'x' as an exception ('x' can be any valid type)
Invoke(f), InvokeWithoutArgs(f)	Call 'f' – with or without the arguments passed to us
Invoke(ptr, &c::m), InvokeWithoutArgs(ptr, &c::m)	As above, but calling the method 'm' belonging to class 'c' on the instance whose address is held in 'ptr'
WithArg<N>(a) WithArgs<…>(a)	Perform action 'a' with a subset of the arguments
DoAll(…)	Perform multiple actions, returning the result of the last

Appendix Two

The GNU Toolchain

garth@ggilmour.com

GNU (GNU's not UNIX)

- In 1983 Richard Stallman launched the GNU Project
  - To write an OS and all the tools a developer needed
  - This became the focus of the 'free software' movement
- GNU tools remain essential
  - The 'gcc' C compiler
  - The 'g++' C++ compiler
  - Sed, Awk, Emacs etc...

The Copyleft Philosophy

- Copyleft is a term coined by GNU and the FSF
  - It means the use of copyright law to ensure that code can be freely distributed
- Consumers of your code are allowed to pass it along
  - This is 'free as in speech'
- There are many (50+) open source licenses available
  - The most famous is the GNU GPL by Richard Stallman

The GNU General Public License

- The GPL allows you to:
  - Use a privately modified version without publishing it
  - Sell copies of a product or charge a fee for downloads
  - Release code under both the GPL and other licenses
  - Use GPL'd tools to create proprietary products
  - Adapt the license for your own use
- The GPL requires that you:
  - Publish the code if you release a modified version
  - Distribute a copy of the GPL with the software
- The only grey area concerns libraries
  - For which there is the 'Lesser GPL' or LGPL

Introducing GCC

- The GNU project was set up to create open source tools
  - Everything needed by a developer should be available for free
- GCC originally meant the 'GNU C Compiler'
  - It was the first open source ANSI C Compiler
    - The original release was made in 1987
  - Without it products like Linux could not have happened
- Over time other compilers were added
  - Including C++, Fortran , ADA and Java
- GCC became the 'GNU Compiler Collection'
  - The C and C++ compilers have become the de facto reference implementations of their targeted languages

Running the GCC Compilers

- The GNU C and C++ compilers have identical options
  - The C compiler is run with 'gcc'
    - Note you may need 'gcc -std=c99' for C99 features
  - The C++ compiler is run with 'g++'
- Multiple files can be compiled into an executable
  - E.g. 'gcc app1.c app2.c -o myapp' in C
  - E.g. 'g++ app1.cpp app2.cpp -o myapp' in C++
  - The '-o' option specified the name of the program
- Note that there is no need to specify header files
  - These are included in the source via '#include'

Running the GCC Compilers

- You don't have to compile the program in a single step
  - Instead you can build object files and link them afterwards
- The '-c' option compiles a source file to an object file
  - E.g. 'gcc -c app1.c' creates 'app1.o'
- If you pass object files to the compiler they are linked
  - E.g. 'gcc app1.o app2.o -o myapp'
  - This is done using either the GNU linker (known as 'ld') or the linker supplied with your operating system
- You want to build the system incrementally
  - Normally using the 'make' tool

Linking Against Libraries

- The GNU compilers search for header files and libraries
  - But only standard directories are searched by default
  - UNIX libraries are found in '/usr/lib' and '/usr/local/lib' and header files in '/usr/include' and '/usr/local/include'
- Libraries are referenced via the '-l' option
  - E.g. 'gcc -labc app1.c' would search for both 'libabc.a' and 'libabc.so' in the standard locations
- The compilers prefer shared over static libraries
  - So 'libabc.so' will be used rather than 'libabc.a'
  - Dynamic linking allows programs to be smaller
    - Plus they can be updated as long as the interface does not change

Locating Additional Libraries

- Extra libraries and header files will not be found
  - If they are placed outside of the standard directories
  - Environment variables can be used to specify these locations
- Three environment variables can be used:
  - 'LIBRARY_PATH' adds extra locations for libraries
  - 'C_INCLUDE_PATH' adds locations for C header files
  - 'CPLUS_INCLUDE_PATH' adds locations for C++ header files
- A further variable is needed for shared libraries
  - 'LD_LIBRARY_PATH' specifies locations were a program can find the '.so' libraries it was compiled against

Locating Additional Libraries

- Many entries can be placed in the environment variables
  - As long as they are separated by colons
- Compiler options can be used instead
  - The '-I' option specifies locations for header files
  - The '-L' option specifies locations for libraries
  - E.g. 'gcc -I/dev/include -L/dev/lib myapp1.c'
- Environment variables and options can be combined
  - In which case directories specified in compiler options are searched before those listed in environment variables

Linking in C and C++

- Translation Units and Libraries need to be linked
  - The linker takes the unmatched symbols within each compiled translation unit (aka object file) and tries to resolve them
- Each library contains a table that lists exported symbols
  - It is possible to generate and add to these tables manually
- In C code the symbol for a function is its name
  - This is possible because C does not support overloading
- In C++ code the symbol for a function is more complex
  - You can disable this by declaring a function as 'extern "C"'
  - This enables a C program to call functions in a C++ library

Linkage and Name Mangling

- C++ compilers use 'Name Mangling'
  - The entries in the exported symbols table are an encoded representation of the signature
  - For example 'func(unsigned int, const double, float)' might be represented as 'func_FUiCdf'
- 'Name Mangling' has several benefits
  - Function overloading can be fully supported
  - Nested types & namespaces don't produce ambiguous symbols
  - User defined symbols don't clash with compiler generated ones
- Historically there were issues with space and efficiency
  - Mangled names can be very long and take time to read

Name Mangling Symbols

Mangling Conventions Part I	
Type	Letter
Void	v
Char	c
Short	s
Int	i
Long	l
Float	f
Double	d
Long double	r
Variable Args	e

Mangling Conventions Part II	
Type	Letter
Unsigned	U
Const	C
Volatile	V
Signed	S
Pointer	P
Reference	R
Array of length n	An_
Function	F
Pointer to nth member	MnS

Using the GNU Debugger (GDB)

- GDB lets you debug C and C++ applications
  - There is partial support for other languages as well
  - It is a command line driven application (no GUI)
- There are three standard ways of using GDB
  - 'gdb *program*' loads an executable to run within GDB
    - Nothing happens until you issue the 'run' command
    - You enter input as required till you get to the problem area
  - 'gdb *program core*' loads state from the core file
    - You can start debugging where the error occurred
  - 'gdb *program pid*' attaches it to a running program
    - The process is immediately stopped so you can check the current state, set breakpoints and then resume execution

Using the GNU Debugger (GDB)

- The object files and executables you run under GDB must contain debugging information
  - This can be thought of as a table which associates blocks of assembly instructions with line numbers in source files
  - This is done in GCC via the '-g' option
- You cant debug a core file without the executable
  - Because the 'symbol table' lives inside the executable
  - The core file is usually in the current directory but:
    - Some shells can be configured not to write core files
    - Some versions of UNIX (including Solaris) can be configured to store cores in a specific directory, such as '/var/coredumps'

Some Essential GNU Commands

- The most essential commands are:
  - 's' and 'n' to step through your code
    - 's' will enter any functions called on the current line
  - 'l' to print blocks of code around the current position
    - By default the next 10 lines are displayed
  - 'b' to set breakpoints in the code
    - Defined via function names, file names and line numbers
- GDB supports TAB completion
  - E.g. if you wanted to set a breakpoint on a function starting with 'set_' you could type 'b set_ *TAB*' to see the options
  - Quoting may be required if you need to use parenthesis
    - Such as when working with function overloading in C++

Command	Description
run *arguments*	Run the program with optional arguments
bt bt *number_of_frames*	Backtrace i.e. display the current position on the call stack
p *expression*	Print the value of the supplied expression
n	Move to the next line, ignoring function calls
s	Move to the next line, stepping into function calls
b *file:function*	Set a breakpoint on the specified function and optionally file
b *file:line*	Set a breakpoint on the specified line and optionally file
b *+offset* b *-offset*	Set a breakpoint 'n' lines before or after the current position
info break	Show all breakpoints
c	Continue running after a breakpoint
finish	Run to the end of the current stack frame
clear	Clear all breakpoints
clear *file:function* *clear file:line*	Clear the specified breakpoint

Command	Description
info func *regex*	Display information on all functions matching 'regex'
whatis *expr* ptype *expr*	Show basic and detailed information about the type of 'expr'
info source	Show name of current source file
info sources	Show the names of all the source files in use
l list	Show the next 10 lines of source code
list -	Show the previous 10 lines of source code
list *x, y*	Show all the source code between lines 'x' and 'y'
list *x*	Show lines centered around 'x'
set var *x=y* set variable *x=y*	Change the value of a variable
help *command*	Get information about how to use *command*
pwd cd *path*	Show and change the current working directory

Breakpoints in More Detail

- Numbers are assigned to breakpoints as you set them
  - This is for convenience for use in later commands
  - E.g. you can refer to breakpoint eight as '$8'
- GDB supports two extra types of breakpoint:
  - A watchpoint stops your program when a value changes
    - The value can be specified via a variable name, a location in memory or the result of evaluating an expression like 'a+b-c'
    - Watchpoints can be implemented in hardware or software
      - The latter may slow down the execution of your program considerably
  - A catchpoint stops your program on an event
    - Such as an exception being thrown in C++ code

Working with the Call Stack

- Your program is always stopped on a stack frame
  - A frame is added to the 'call stack' on each function call
  - The first function on the call stack is always 'main'
    - This is referred to as the 'initial' or 'outermost' frame
    - The most recent or 'innermost' frame is where code is running
  - GDB assigns numbers to stack frames starting at 0
- The 'backtrace' / 'bt' command displays the call stack
  - You can limit the number of frames displayed as required
- The 'frame' command changes the current frame
  - Allowing you to print the value of local variables there
  - You can also use the 'up' and 'down' commands

Introducing Make

- Make is a tool for automatically building targets
  - The targets will typically be C/C++ executables and libraries
- Make builds targets based on dependencies
  - Each target is associated with one or more dependencies
  - The dependencies may themselves be targets
    - Any amount of nesting is supported
- A build tool like Make is crucial for developers
  - Several variants of Make are available (e.g. Boost.Jam)
  - Ant / Maven, NAnt / MSBuild and Rake / Raven are used by Java, C# and Ruby developers respectively

The Structure of a Makefile

- A makefile is make up of rules
  - The first rule in the file is the 'default rule'
  - This is the only rule Make runs by default
- All rules have the same basic structure
  - The name of the target comes first, followed by a colon
  - Then the dependencies, followed by a new line
  - Finally zero or more commands, each on its own line
- Commands must be prefixed with a TAB
  - Not multiple spaces or any other whitespace characters
  - This is a very common source of mistakes

A 'Hello World' Makefile

- Consider a simple C++ program
  - We have a file with a 'main' method ('main.cpp')
  - We have a file containing maths functions ('maths.cpp')
  - The prototypes for the maths functions are placed in a separate header file ('maths.h')
- We need Make to do the following:
  - Compile each of the '.cpp' files into an object file
    - Files should only be compiled if required
  - Link the object files together into an executable
  - Remove all generated files when required

maths.h

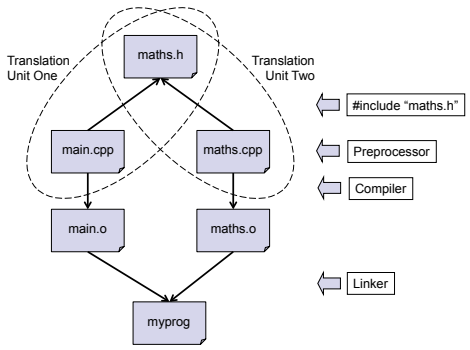
```
//function declarations (prototypes)
int add(int no1, int no2);
int subtract(int no1, int no2);
double circumference(double diameter);
```

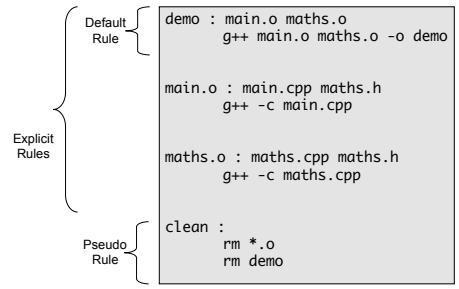
main.cpp (C++ Version)

```
#include "maths.h"
int main() {
    cout << "Addition gives "
         << math::add(12,14)
         << endl;
    cout << "Subtraction gives "
         << math::subtract(30,18)
         << endl;
    cout << "Circumference is "
         << circumference(10)
         << endl;
}
```

maths.cpp

```
#include "maths.h"
int add(int no1, int no2) {
    return no1 + no2;
}
int subtract(int no1, int no2) {
    return no1 - no2;
}
double
circumference(double diameter) {
    return PI * diameter;
}
```




```
> ls
main.cpp makefile maths.cpp maths.h
> make
g++ -c main.cpp
g++ -c maths.cpp
g++ main.o maths.o -o demo
> ls
demo main.cpp main.o makefile maths.cpp maths.h maths.o
> touch maths.cpp
> make
g++ -c maths.cpp
g++ main.o maths.o -o demo
> ls
demo main.cpp main.o makefile maths.cpp maths.h maths.o
> make clean
rm *.o
rm demo
> ls
main.cpp makefile maths.cpp maths.h
```


How Make Executes

- Make first determines which target to build
  - This is either the default or one you have specified
- Nothing is done if the target predates its dependencies
  - Be careful with times when moving files between machines
- Otherwise each dependency is examined
  - If a dependency is a target, and is older than one or more of its own dependencies then it must be built first
  - This process continues recursively
- Most targets will have a tree of dependencies
  - Note that only out of date parts of the tree are rebuilt

Features of Make

- Make has the following features:
  - Pattern based rules
  - Special targets
  - Implicit rules
  - Variables and Macros
  - Functions
- Most projects build makefiles automatically
  - IDE's like Visual Studio and Eclipse CDT are based on it
  - Many developer power-tools parse a source tree and produce an appropriate makefile (e.g. imake, makedepend)

Appropriate Jobs For Make

- Make is suitable for any job involving dependencies
  - Not just building programs from source code
  - Consider a website where HTML is being generated from XML with hyperlinks and randomly assigned filenames
    - Make can be used to ensure that pages are created in the correct order and hence all links are valid
- There is some support for common tasks
  - Make understands C/C++ naming conventions
  - There is a database of built in (aka implicit) rules
    - To view these use 'make --print-data-base'

Rules in Detail

- The simplest type of rule is an explicit one
  - Where target and dependency names are hard-coded
  - The same target name can be used in many rules
    - Make combines all the dependencies into a single list
- Pattern based rules use wildcards
  - They apply to all files of a particular type
- A rule can have a phony target
  - The target is not an entity to be built, but rather an identifier that triggers an arbitrary set of commands
  - E.g. a 'clean' target that deletes intermediate files

Special Targets

- Some rules can be marked as special
  - By using built in special targets defined by Make
- Consider '.PHONY : clean'
  - This tells Make that 'clean' is a phony rule
  - It prevents errors if a file called 'clean' is put in the project
- Another example is '.INTERMEDIATE'
  - Any dependencies of this rule are intermediate files
  - If Make creates one of these files whilst executing another rule it will delete the file before it exits

Variables in Makefiles

- Variables can be used in makefiles
  - They are both automatic and user defined
  - The syntax for using a variable is '$(*var_name*)'

Variable	Description
$@	The name of the target
$%	The name of the target when the target is an archive (UNIX library)
$<	The name of the first dependency
$?	The names of all dependencies newer than the target
$^	The names of all dependencies excluding duplicates
$+	The names of all dependencies including duplicates
$*	The stem of the target (suffix is removed)

Variable Declaration

Rule Using Variables

Relying on Pattern Rule

Pattern Based Rule

Special Target

Phony Rule (That Make Knows About)

```
object_files = main.o maths.o

demo : $(object_files)
	@echo 'building executable'
	g++ $(object_files) -o $@

main.o : main.cpp maths.h
maths.o : maths.cpp maths.h

%.o : %.cpp
	@echo building object $@
	g++ -c $^

.PHONY : clean

clean :
	rm *.o
	rm demo
```

Commands in Depth

- Make passes each command to a shell
  - You can specify which shell via the 'SHELL' variable
  - Commands can span multiple lines via '\'
- Not everything under the rule is a command
  - Lines starting with a '#' are comments
  - Blank lines are ignored
- Prefixes can be used to modify commands
  - Placing '@' before the command means it is not written
    - So '@echo fred' would only print 'fred'
  - The '-' prefix means errors will be ignored

Commands in Depth

- By default Make only looks in the current directory
  - You can add extra locations via the 'VPATH' variable
  - The locations can be separated by spaces or by the platform default (colons on UNIX and semi-colons on Windows)
- There is also a directive called 'vpath'
  - This is used with a pattern to locate files
  - E.g. 'vpath %.h include' or 'vpath %.class classes'
- Make can be used recursively
  - I.e. you can call Make as a command within a rule
  - This is typically used when one makefile triggers others in sub-folders to build sub-projects

Function Calls In Makefiles

- Make contains its own library of functions
  - The syntax for calling a function in Make is either '$(function arguments)' or '${function arguments}'
- The functions cover a range of categories
  - String manipulation functions such as 'subst'
  - Filename functions such as 'dir' and 'notdir'
  - Flow control functions such as 'if' and 'error'
- The 'eval' function is special
  - It expends the arguments and evaluates then within Make
  - You can use it to add new variables and rules on the fly

Autoconf and Automake

- Writing makefiles by hand can be manageable
  - If the number of files is small enough
  - If new source files are added infrequently
  - If the program runs on a single platform
- Otherwise it is best to generate them
  - Many tools are available for generating makefiles
  - The GNU project provides 'autoconf' and 'automake'
- These tools are difficult to work with
  - Knowing the differences between OS's is a task in itself
  - Normally a 'build master' takes responsibility for generating a build file for each supported platform

Autoconf and Automake

- Portability is a problem even across UNIX variants
  - Header files and functions may have different names
  - Functions may be in different header files, have different signatures or simply not be supported
- This is especially true for library vendors
  - Who have to support all versions used by clients
- Autoconf and Automake help manage this for you
  - The input to Autoconf is a file called 'configure.ac'
    - In earlier versions this was called 'configure.in'
  - The input to Automake is a file called 'Makefile.am'
  - The output is a build file specific to the current OS

Understanding Autoconf

- The 'configure.in' file is a bourne shell script
  - It contains assertions about files, functions etc…
- The assertions are made in the form of macros
  - A separate tool, known as the 'GNU m4 macro preprocessor', defines and expands the macros
  - E.g. 'AC_CHECK_FILE(*p*,*s1*,*s2*)' looks for the file at *p*
    - Running the shell code *s1* if it can be found and *s2* otherwise
- Macros are used to fix as well as find problems
  - E.g. 'AC_REPLACE_FUNCS' looks for a list of functions
    - If 'foo' can be found then a 'HAVE_FOO' symbol is defined
    - Otherwise it will look for and include 'foo.o' in the build

Understanding Automake

- The 'Makefile.am' file lists the parts of the program
  - These are defined using variables known as primaries
  - The 'HEADERS' variable is a list of header files
  - The 'SOURCES' variable is a list of source files
  - The 'DATA' variable is a list of resources
    - Items such as configuration and help files
- Automake uses this to generate a makefile template
  - This is called 'Makefile.in' and contains targets
    - Which have familiar names such as 'clean', 'dist' and 'install'
  - Autoconf can take this file and combine it with what it already knows to generate an OS specific buildfile

The Process of Creating a Makefile

1. Create the 'configure.ac' configuration file
   - There is an 'autoscan' script available to get you started
2. Create the 'Makefile.am' description file
3. Run the 'aclocal' script
   - This creates platform specific versions of the macros using by Autoconf and places them in a file called 'aclocal.m4'
4. Run Automake to generate 'Makefile.in'
   - You will need to have files common to all GNU projects in the home directory (e.g. 'ChangeLog' and 'README')
5. Run Autoconf to generate a script called 'configure'
6. Execute 'configure' to create the OS specific makefile

Appendix Three

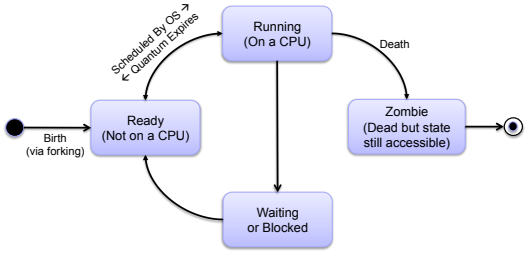
Linux Concurrency Basics

garth@ggilmour.com

The UNIX Process Model

- A process is an instance of an executable file
  - Managing (or scheduling) processes is the most important task performed by the Operating System
  - Information about each process is stored in a list and they are allocated runtime based on relative priorities
- Processes are created via forking
  - The kernel copies a process and adds the copy to the list
  - The child is initially given half or its parents remaining CPU time
- Each process is assigned a unique identifier (PID)
  - These are assigned in sequence until a maximum value is reached, after which the lowest available PID is reused

The States of a UNIX Process



The Structure of a Process

- A process is made up of:
  - The 'Text Segment' holding instructions
  - The 'Data Segment' holding statically allocated variables
  - The 'Heap Space' for dynamically allocated variables
  - The 'Call Stack' for managing chains of function calls
    - Parameters and local variables live on frames of the stack
- A running process has a context of execution
  - This includes the current value in the program counter and any values stored in registers or other memory buffers
  - At any given time there can only be one context per CPU

The UNIX Process Model

- Processes work in kernel mode and user mode
  - Use mode is when a process executes its own code
  - Kernel mode is when the process is running OS code
- Kernel mode is entered when:
  - The process calls a function declared in the system API's
  - The process calls a function that accesses a device
  - An exception occurs or an interrupt arrives
- A process may contain multiple threads
  - This may be an illusion depending on the OS version
  - All threads in the process share the same address space

Processes and Signals

- Signals let you send messages between processes
  - You can send them via the keyboard or the 'kill' command
- A program can handle a signal any way it likes
  - With the exception of the KILL and STOP signals
  - These always result in the process being shut down

Signal	Number	Meaning
HUP	1	Hangup – used with modems
INT	2	Interrupt (CTRL-c)
QUIT	3	Stop running and dump code (CTRL-\)
KILL	9	Unconditional and permanent stop
STOP	17	Stop unconditionally till told to continue
CONT	19	Tells a stopped process to continue

Processes and Interrupts

- An interrupt is an asynchronous event
  - It is delivered by a device called an interrupt controller
  - The CPU could be at any point in its execution cycle
- Interrupts are essential to the OS
  - They enable the kernel to manage multiple devices
- The System Timer generates interrupts
  - It counts down from a specified value to zero
  - The interrupt is generates is known as a 'tick'
- Processes run for a certain number of 'ticks'
  - This is known as the 'quantum' or 'timeslice'
  - In Linux this value can vary (see later)

Comparing Processes and Threads

- Processes and threads are very different things
  - A process is a memory space which by itself is inert
  - A thread is a separate line of execution
    - It is made up of one or more call stacks
- Threads need to be scheduled onto processors
  - A thread is allocated a pre-defined time to run
    - This is known as the 'timeslice' or 'quantum'
  - When this time has expired a 'context switch' occurs
    - All values stored on registers are captured and stored
    - The values associated with another thread are loaded

Concurrency in Linux and Windows

- Unix and Windows approach concurrency differently
  - Windows threads are lightweight entities and developers are advised to use them rather than start new processes
  - On UNIX the focus has traditionally been on processes
- The 'LinuxThreads' library started to reverse this trend
  - But threads were actually kernel processes in a 1-to-1 mapping, with a 'manager thread' maintaining the illusion
- This was replaced with 'Native POSIX Thread Library'
  - NPTL still uses a 1-to-1 mapping with kernel tasks

Introducing the 'PThreads' Library

- Most UNIX threading is based around 'PThreads'
  - 'POSIX Threads' provides a standard threading API
    - Enabling you to easily port code across platforms
  - It is available on most operating systems
    - Including Microsoft Windows and Apple OS X
- The 'PThreads' library provides support for:
  - Starting threads based on an arbitrary function
    - Note that the function must take and return a 'void *'
  - Yielding, cancelling and signalling between threads
  - Synchronizing shared access to data
    - Using mutexes, conditions and read/write locks

Linux Thread Scheduling

- Linux has used a variety of scheduling models
  - The latest is the 'Completely Fair Scheduler' (CFS)
    - Adopted in version 2.6.23 of the kernel
  - Note that the scheduler architecture is modular
    - Different 'scheduler classes' can be plugged in as needed
    - CFS is registered for 'SCHED_NORMAL' processes
- CFS gets its name by handing out CPU time fairly
  - If there are 10 processes running then, over any given period, each process should receive 10% of the CPU time
  - This is accomplished via variable timeslices
    - Unlike Windows, where the duration is fixed, the length of the next timeslice is calculated based on the count of processes

Linux Thread Scheduling

- Picking the next process is also based on fairness
  - An I/O bound program like a GUI will spend most of its time waiting on keyboard and mouse events
  - So when an event fires and it enters the runnable state CFS will note that it is behind on its 'fair share'
    - Hence it will be highly likely to be picked next
    - Processes which are CPU bound will get their 'fair share' of processor time in the periods between I/O events occurring
- The CFS organises processes in a balanced tree
  - As opposed to the list used for 'round-robin' scheduling
  - This ensures it can find a process as fast as possible

Linux Thread Scheduling

- A process can specify a 'nice value'
  - This lies between -20 and 19 (the default is 0)
  - Higher values result in the process having a lower priority
    - And hence a share of CPU time less than '1/N'
  - Nice values alter the relative lengths of timeslices
    - Because the process is being 'nice' to others
- A timeslice must exceed the 'minimum granularity'
  - This is due to the overhead imposed by context switching
  - Without this the system could burn itself out
    - In a situation with 1000's of processors the CFS would generate tiny timeslices in an attempt to be fair, resulting in the majority of CPU time being spent switching contexts on and off the CPU

Scheduling and Preemption

- Scheduling will occur when:
  - The timeslice of the current process has expired
    - The System Timer has sent the correct number of 'ticks'
  - A more deserving process has become runnable
  - A process is moving from kernel mode to user mode
    - Often because a system call has completed
    - This is known as 'user preemption'
- One kernel task can also preempt another
  - When a process enters kernel mode without holding locks
  - When a kernel task blocks or yields the rest of its timeslice

Scheduling and Processor Affinity

- Processor affinity occurs when a scheduler attempts to repeatedly run a thread on the same CPU
  - This increases performance because it lowers the complexity of context switching at the hardware level
- Both Windows and Linux provide 'soft affinity'
  - By default they try to keep a thread on one CPU
  - On Windows this is referred to as its 'ideal processor'
- They also allow a user to specify 'hard affinity'
  - Code can insist that a thread live on a specified CPU
  - This is discouraged when writing conventional applications
    - It can be used to give a program an unfair performance boost

Inter-Process Communication

- Multi-process applications are the default in Linux
  - As we have seen multi-threading is just a special case
- Processes within the same app must exchange data
  - But a process by definition has a separate address space
- Like other OS's Linux offers an IPC library
  - 'Inter-Process Communication' refers to the set of mechanisms by which processes can exchange data
- IPC and locking are closely related
  - Although locks are not used to convey information they allow turn-about access to the IPC mechanism in use
  - Semaphores can be used for very simple communications

IPC with Pipes and FIFO's

- All UNIX users know Pipes and FIFO's
  - A pipe is a unidirectional flow of data
    - This what you get when you say 'ls –al | more'
  - Pipes are mostly used between parents and children
    - The parent process creates the pipe and then forks
    - The child process closes the write end of the pipe and waits while the parent closes the read end and writes (or vice versa)
- A FIFO is also known as a 'Named PIPE'
  - A regular pipe cannot be used by unrelated processes
    - Unless they pass a descriptor via another IPC mechanism
  - A FIFO is created based on an regular UNIX pathname
    - This can be done in code or via the 'mkfifo' command

IPC with Message Queues

- A message queue is a list of records
  - Each queue has 'kernel persistence' meaning that it lasts until it is explicitly closed or the OS is restarted
  - A process can write some messages and exit, knowing that another process will read them at a later time
- The sender can assign a message a priority
  - On Linux this is an integer in the range 0 to 32768
- POSIX queues support asynchronous events
  - Listeners can be notified when a message is ready
    - Without needing to block on the queue or perform polling
  - POSIX style queues were introduced in kernel 2.6.6

IPC with Shared Memory

- Shared memory is the fastest form of IPC
  - The kernel maps a file into multiple processes
    - The file does not have to start at the same address
      - If you pass around offsets into the file they must be relative
    - Parent / child processes can map anonymous memory instead of a named file (introduced in kernel version 2.4)
  - The processes can then use the memory as normal
    - Once the memory is mapped the kernel is no longer involved
    - There are issues with addressing into very large files
- Synchronization is usually required
  - To ensure that reads and writes occur in the correct order
  - Any of the standard lock types can be used for this

Creating Multiple Processes

- New processes are created via 'fork'
  - This causes the kernel to make an identical copy of the current process and add it to the process tree
    - Memory pages are copied only when required for efficiency
  - You can tell which process you are in via the return value
    - In the parent process 'fork' returns the process ID of the child
    - In the child process 'fork' returns 0
      - If necessary the child can use 'getpid' to discover its own id
- Normally the parent waits for the child to complete
  - It collects the return value and the child process dies
  - Otherwise when the child exits it remains as a 'zombie'


```
using std::cout;
using std::endl;

void func(const char * name);

const int SLEEP_IN_SECONDS = 1;
const int ITERATION_COUNT = 20;
const int NUM_PROCESSES = 4;

int main() {
	const char * processNames[] = {"Process A", "Process B", "Process C", "Process D"};
	pid_t processIDs[NUM_PROCESSES];
	int exitStatusValues[NUM_PROCESSES];

	for(int i=0;i<NUM_PROCESSES;i++) {
		if((processIDs[i] = fork()) == 0) {
			//we are in a child process
			func(processNames[i]);
			exit(0);
		} else {
			printf("Just started %s with id %d\n",processNames[i],processIDs[i]);
		}
	}
```



```
	cout << "Waiting for processes to complete";
	for(int i=0;i<NUM_PROCESSES;i++) {
		waitpid(processIDs[i],&exitStatusValues[i],0);
	}

	cout << "Processes exited with values ";
	for(int i=0;i<NUM_PROCESSES;i++) {
		cout << exitStatusValues[i] << " ";
	}
	cout << endl;

	cout << "End of main" << endl;
}
void func(const char * name) {
	for(int i=0;i<ITERATION_COUNT;i++) {
		printf("%s message %d\n",name,i);
		sleep(SLEEP_IN_SECONDS);
	}
}
```

Sharing Data Via Pipes

- Related processes typically share data via pipes
  - Unrelated processes can use named pipes (aka FIFO's)
  - You can create them via the 'pipe' function
    - This takes the address of an array of two file descriptors
    - The first is set up for reading and the second for writing
- 'popen' and 'pclose' simplify pipes further
  - 'popen' creates a child process which will execute the UNIX command you specify via the default shell
  - It returns a pipe that you can use to either read or write
    - You specify which via the second argument ("r" or "w")
  - 'pclose' waits for the child to exit and closes the pipe


```
void writeData(FILE * fp);

int main() {
	const char * command1 = "egrep [A-Z]{3}";
	const char * command2 = "egrep [0-9]{3}";
	const char * command3 = "egrep ghi$";

	FILE * fp1 = popen(command1, "w");
	FILE * fp2 = popen(command2, "w");
	FILE * fp3 = popen(command3, "w");

	printf("### About to send input to '%s' ###\n", command1);
	writeData(fp1);
	pclose(fp1);

	printf("### About to send input to '%s' ###\n", command2);
	writeData(fp2);
	pclose(fp2);

	printf("### About to send input to '%s' ###\n", command3);
	writeData(fp3);
	pclose(fp3);
}
```

```
void writeData(FILE * fp) {
	fprintf(fp,"abc def 123\n");
	fprintf(fp,"abc DEF ghi\n");
	fprintf(fp,"abc 456 ghi\n");
	fprintf(fp,"789 def GHI\n");
}
```



```
int main() {
	pid_t childProcessID;
	int fileDescriptors[2];
	int exitStatus;

	pipe(fileDescriptors);
	if((childProcessID = fork()) == 0) {
		close(fileDescriptors[1]);
		char buffer[10];
		for(int i=0;i<10;i++) {
			read(fileDescriptors[0],buffer,10);
			cout << "Child process (" << getpid() << ") just read in " << buffer << endl;
		}
		exit(0);
	} else {
		close(fileDescriptors[0]);
		char buffer[10];
		for(int i=0;i<10;i++) {
			sprintf(buffer,"Message %d",i);
			write(fileDescriptors[1],buffer,10);
			sleep(2);
		}
	}
	waitpid(childProcessID,&exitStatus,0);
	cout << "Child process exited with value " << exitStatus << endl;
}
```

Creating Multiple Threads

- You create new threads via 'pthread_create'
  - The first parameter is the address of a handle
    - This of type 'pthread_t' and identifies the thread in any other call
  - The second is the address of a 'pthread_attr_t'
    - This customizes threading attributes, pass 0 to get the defaults
  - The third is the address of a function
    - This will be used as the 'main' of the new call stack
  - The fourth is data to be passed into the function
    - When it is being launched as the start of a new thread
- The function you use must take and return a 'void *'
  - This is to let you pass in and get back arbitrary data

Creating Multiple Threads

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <iostream>
#include <string>

using namespace std;

const int WAIT_IN_MICROSECONDS = 250000;
const int ITERATION_COUNT = 50;

void * func(void * data) {
	const char * name = (char *)data;
	for(int i=0;i<ITERATION_COUNT;i++) {
		printf("%s message %d\n",name,i);
		usleep(WAIT_IN_MICROSECONDS);
	}
	string * retval = new string(name);
	retval->append(" finished!");
	return retval;
}
```



```
int main() {
	pthread_t threadOne;
	pthread_t threadTwo;
	pthread_t threadThree;

	pthread_create(&threadOne, 0, func, (void *)"first thread");
	pthread_create(&threadTwo, 0, func, (void *)"second thread");
	pthread_create(&threadThree, 0, func, (void *)"third thread");

	string * val1;
	pthread_join(threadOne,(void **)&val1);
	string * val2;
	pthread_join(threadTwo,(void **)&val2);
	string * val3;
	pthread_join(threadThree,(void **)&val3);

	cout << "All threads completed with return values:" << endl;
	cout << "\t" << *val1 << endl;
	cout << "\t" << *val2 << endl;
	cout << "\t" << *val3 << endl;

	delete val1;
	delete val2;
	delete val3;
	cout << "End of main" << endl;
}
```

Working With Mutexes

- A mutex is of type ‘pthread_mutex_t’
  - Normally these are created from heap memory and initialized via a call to ‘pthread_mutex_init’
  - You can optionally pass in a ‘pthread_mutexattr_t’
    - This allows you to create (for example) recursive mutexes
- There are two ways to acquire a mutex
  - ‘pthread_mutex_lock’ waits for it to become available
  - ‘pthread_mutex_trylock’ tests and returns immediately
    - Allowing you to perform other work till the lock is available
- A mutex is released via ‘pthread_mutex_unlock’
  - Forgetting to release a lock is a very common error

Waiting on a Mutex

```
struct MyData {
	const char * threadName;
	pthread_mutex_t * theMutex;
};
void * func(void * data) {
	MyData * realData = (MyData *)data;
	for(int i=0;i<ITERATION_COUNT;i++) {
		pthread_mutex_lock(realData->theMutex);
		printf("%s iteration %d message A\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
		printf("%s iteration %d message B\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
		printf("%s iteration %d message C\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
		printf("%s iteration %d message D\n",realData->threadName,i);
		pthread_mutex_unlock(realData->theMutex);
		usleep(WAIT_IN_MICROSECONDS);
	}
	return 0;
}
```



```
int main() {
	pthread_t threadOne;
	pthread_t threadTwo;
	pthread_t threadThree;

	pthread_mutex_t * mutex = new pthread_mutex_t;
	pthread_mutex_init(mutex,0);

	MyData data1 = {"first thread", mutex};
	MyData data2 = {"second thread", mutex};
	MyData data3 = {"third thread", mutex};

	pthread_create(&threadOne, 0, func, (void *)&data1);
	pthread_create(&threadTwo, 0, func, (void *)&data2);
	pthread_create(&threadThree, 0, func, (void *)&data3);

	pthread_join(threadOne,0);
	pthread_join(threadTwo,0);
	pthread_join(threadThree,0);

	cout << "All threads completed" << endl;
	pthread_mutex_destroy(mutex);
	cout << "End of main" << endl;
}
```

Spinning On a Mutex

```
int main() {
	/* Beginning of main identical to previous demo...  */

	usleep(LONG_WAIT_IN_MICROSECONDS); //let the threads start
	while(pthread_mutex_trylock(mutex)) {
		printf("Main thread cant acquire mutex - spinning...\n");
		usleep(LONG_WAIT_IN_MICROSECONDS);
	}
	printf("Main thread has acquired mutex\n");
	pthread_mutex_unlock(mutex);

	pthread_join(threadOne,0);
	pthread_join(threadTwo,0);
	pthread_join(threadThree,0);

	cout << "All threads completed" << endl;
	pthread_mutex_destroy(mutex);
	cout << "End of main" << endl;
}
```


Working with Conditions

- A common threading pattern is as follows:
  - A thread acquires a mutex and tries to use some data
  - If the data isn't there it must release the mutex and block till another thread has created and stored the data
    - This second thread will send a signal when it is done
  - When the signal is received the original thread must unblock, reacquire the mutex and process the data
- Various race conditions lurk within this pattern
  - E.g. what happens if the signal is sent *between* the original thread releasing the mutex and starting to block?
- Conditions are a solution to this problem

Working with Conditions

- A condition is created just like a mutex
  - You create a variable of type 'pthread_cond_t' and use 'pthread_cond_init' to set its intitial state
- The most important function is 'pthread_cond_wait'
  - This accepts the addresses of a mutex and a condition
  - The function causes the calling thread to release the mutex and start to wait on the condition as a single action
  - When woken the thread automatically reacquires the lock
- Waiting threads can be signalled in two ways
  - 'pthread_cond_signal' wakes one thread
  - 'pthread_cond_broadcast' wakes up all threads

Waiting on a Condition

```
struct MyData {
	const char * threadName;
	pthread_mutex_t * theMutex;
	pthread_cond_t  * theCondition;
};

void * func(void * data) {
	MyData * realData = (MyData *)data;
	pthread_mutex_lock(realData->theMutex);

	printf("%s about to start waiting\n",realData->threadName);
	pthread_cond_wait(realData->theCondition, realData->theMutex);
	printf("%s has been woken up\n",realData->threadName);
	pthread_mutex_unlock(realData->theMutex);

	for(int i=0;i<ITERATION_COUNT;i++) {
		pthread_mutex_lock(realData->theMutex);
		printf("%s iteration %d message A\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
```



```
		printf("%s iteration %d message B\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
		printf("%s iteration %d message C\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
		printf("%s iteration %d message D\n",realData->threadName,i);
		pthread_mutex_unlock(realData->theMutex);
		usleep(WAIT_IN_MICROSECONDS);
	}
	return 0;
}
int main() {
	pthread_t threadOne;
	pthread_t threadTwo;
	pthread_t threadThree;

	pthread_mutex_t * mutex = new pthread_mutex_t;
	pthread_mutex_init(mutex,0);

	pthread_cond_t * condition = new pthread_cond_t;
	pthread_cond_init(condition,0);

	MyData data1 = {"first thread", mutex, condition};
	MyData data2 = {"second thread", mutex, condition};
	MyData data3 = {"third thread", mutex, condition};
```



```
	pthread_create(&threadOne, 0, func, (void *)&data1);
	pthread_create(&threadTwo, 0, func, (void *)&data2);
	pthread_create(&threadThree, 0, func, (void *)&data3);

	usleep(LONG_WAIT_IN_MICROSECONDS); //let the threads start
	for(int i=0;i<NUM_THREADS;i++) {
		usleep(LONG_WAIT_IN_MICROSECONDS);
		printf("Main thread about to wake up a thread...\n");
		pthread_cond_signal(condition);
	}

	pthread_join(threadOne,0);
	pthread_join(threadTwo,0);
	pthread_join(threadThree,0);

	cout << "All threads completed" << endl;
	pthread_mutex_destroy(mutex);
	pthread_cond_destroy(condition);
	cout << "End of main" << endl;
}
```

Conditions and Broadcasting

```
int main() {
    /* Beginning of main identical to previous demo... */

    pthread_create(&threadOne, 0, func, (void *)&data1);
    pthread_create(&threadTwo, 0, func, (void *)&data2);
    pthread_create(&threadThree, 0, func, (void *)&data3);

    //let the threads start and enter the waiting state
    usleep(LONG_WAIT_IN_MICROSECONDS);
    printf("Main thread about to wake up all threads...\n");
    pthread_cond_broadcast(condition);

    pthread_join(threadOne,0);
    pthread_join(threadTwo,0);
    pthread_join(threadThree,0);

    cout << "All threads completed" << endl;
    pthread_mutex_destroy(mutex);
    pthread_cond_destroy(condition);
    cout << "End of main" << endl;
}
```


Working with Read / Write Locks

- Sometimes exclusive locking is overkill
  - Consider a data structure that is mostly read-only
  - Any number of readers should be able to share access
    - As long as a writer is not currently modifying the structure
  - Only a single writer should be allowed access at a time
    - If any readers have access then the writer must block
- PThreads supports this pattern via ‘pthread_rwlock_t’
  - Readers acquire the lock via ‘pthread_rwlock_rdlock’
  - Writers acquire the lock via ‘pthread_rwlock_wrlock’
  - Both types use ‘pthread_rwlock_unlock’ to release the lock
    - There must be no locking of any kind for a writer to grab the lock

Working with Read / Write Locks

```
struct MyData {
    const char * threadName;
    pthread_rwlock_t  * theRWLock;
};
void * readerFunc(void * data) {
    MyData * realData = (MyData *)data;

    for(int i=0;i<ITERATION_COUNT;i++) {
        pthread_rwlock_rdlock(realData->theRWLock);
        printf("%s iteration %d message A\n",realData->threadName,i);
        usleep(WAIT_IN_MICROSECONDS);
        printf("%s iteration %d message B\n",realData->threadName,i);
        usleep(WAIT_IN_MICROSECONDS);
        printf("%s iteration %d message C\n",realData->threadName,i);
        usleep(WAIT_IN_MICROSECONDS);
        printf("%s iteration %d message D\n",realData->threadName,i);
        pthread_rwlock_unlock(realData->theRWLock);
        usleep(WAIT_IN_MICROSECONDS);
    }
    return 0;
}
```

```
void * writerFunc(void * data) {
	MyData * realData = (MyData *)data;
	for(int i=0;i<ITERATION_COUNT;i++) {
		pthread_rwlock_wrlock(realData->theRWLock);
		printf("%s iteration %d message A\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
		printf("%s iteration %d message B\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
		printf("%s iteration %d message C\n",realData->threadName,i);
		usleep(WAIT_IN_MICROSECONDS);
		printf("%s iteration %d message D\n",realData->threadName,i);
		pthread_rwlock_unlock(realData->theRWLock);
		usleep(WAIT_IN_MICROSECONDS);
	}
	return 0;
}
int main() {
	pthread_t threadOne;
	pthread_t threadTwo;
	pthread_t threadThree;
	pthread_t threadFour;
	pthread_t threadFive;
	pthread_t threadSix;

	pthread_rwlock_t * rwLock = new pthread_rwlock_t;
	pthread_rwlock_init(rwLock,0);
```



```
	MyData data1 = {"reader thread one", rwLock};
	MyData data2 = {"reader thread two", rwLock};
	MyData data3 = {"reader thread three", rwLock};
	MyData data4 = {"reader thread four", rwLock};
	MyData data5 = {"writer thread one", rwLock};
	MyData data6 = {"writer thread two", rwLock};

	pthread_create(&threadOne, 0, readerFunc, (void *)&data1);
	pthread_create(&threadTwo, 0, readerFunc, (void *)&data2);
	pthread_create(&threadThree, 0, readerFunc, (void *)&data3);
	pthread_create(&threadFour, 0, readerFunc, (void *)&data4);
	pthread_create(&threadFive, 0, writerFunc, (void *)&data5);
	pthread_create(&threadSix, 0, writerFunc, (void *)&data6);

	pthread_join(threadOne,0);
	pthread_join(threadTwo,0);
	pthread_join(threadThree,0);
	pthread_join(threadFour,0);
	pthread_join(threadFive,0);
	pthread_join(threadSix,0);

	cout << "All threads completed" << endl;
	pthread_rwlock_destroy(rwLock);
	cout << "End of main" << endl;
}
```


Working with Shared Memory

- A file is mapped into a process via ‘mmap’
  - The first parameter is the preferred address
    - This is only a hint and is usually left as null
  - The second is the number of bytes to map in
  - The third is the permissions the process will have
  - The fourth is special attributes for the mapping
    - You must specify ‘MAP_SHARED’ for changes to be visible
  - The fifth is a file descriptor identifying the file
  - The sixth is the offset at which mapping should begin
- The memory mapping can be anonymous
  - By using ‘MAP_ANON’ and a file descriptor of -1

```
struct MyData {
        pid_t pid;
        int val1;
        double val2;
        bool val3;
};

int main() {
        //this file could be created via 'dd if=/dev/zero of=test.txt bs=256 count=10'
        const char * path = "/home/ggilmour/tmp/test.txt";
        pid_t firstChildPID;
        pid_t secondChildPID;
        int fileDescriptors[2];
        int exitStatusOne;
        int exitStatusTwo;

        cout << "Main process started with pid " << getpid() << endl;
        int fd = open(path,O_RDWR);
```



```
        void * address = mmap(0,1024,PROT_READ | PROT_WRITE,
                                MAP_SHARED,fd,0);

        if((firstChildPID = fork()) == 0) {
                MyData data = {getpid(), 12,3.4,true};
                MyData * tmp = (MyData *)address;
                *tmp = data;
                exit(0);
        }
        if((secondChildPID = fork()) == 0) {
                MyData data = {getpid(), 57,7.8,false};
                MyData * tmp = (MyData *)address;
                tmp++;
                *tmp = data;
                exit(0);
        }
        waitpid(firstChildPID,&exitStatusOne,0);
        waitpid(secondChildPID,&exitStatusTwo,0);
```



```
        cout << "Child processes exited with values " << exitStatusOne
             << " and " << exitStatusTwo << endl << endl;

        cout << "Data in shared memory is:" << endl;
        MyData * ptr = (MyData *)address;

        cout  << "\t" << ptr->pid << " " << ptr->val1 << " "
              << ptr->val2 << " " << ptr->val3 << endl;

        ptr++;

        cout  << "\t" << ptr->pid << " " << ptr->val1 << " "
              << ptr->val2 << " " << ptr->val3 << endl << endl;

        cout << "End of main process" << endl;
}
```

Shared Memory and Mutexes

- Locks can be created inside shared memory
  - Allowing multiple processes to co-ordinate access
- PThread mutexes and conditions can be used
  - As long as the 'PTHREAD_PROCESS_SHARED' attribute is specified when the mutex or condition is initialized
- Achieving this is somewhat convoluted
  - Create a data structure to represent the attributes
    - This will be a 'pthread_mutexattr_t' or 'pthread_condattr_t'
  - Call the appropriate function to initialize the structure
  - Call the accessor method to set the shared attribute
  - Pass the data into the lock initialization method


```
struct MyData {
	pthread_mutex_t mutex;
	pthread_cond_t condition;
	int data;
};
int main() {
	//this file could be created via 'dd if=/dev/zero of=test.txt bs=256 count=10'
	const char * path = "/home/ggilmour/tmp/test.txt";
	pid_t firstChildPID;
	pid_t secondChildPID;
	int fileDescriptors[2];
	int exitStatusOne;
	int exitStatusTwo;

	cout << "Main process started with pid " << getpid() << endl;

	int fd = open(path,O_RDWR);
	void * address = mmap(0,1024,PROT_READ | PROT_WRITE,
				MAP_SHARED,fd,0);
```



```
	MyData * dataPtr = (MyData *)address;

	//initialize the mutex so it can be used across processes
	pthread_mutexattr_t mutexAttributes;
	pthread_mutexattr_init(&mutexAttributes);
	pthread_mutexattr_setpshared(&mutexAttributes,
					PTHREAD_PROCESS_SHARED);
	pthread_mutex_init(&dataPtr->mutex,&mutexAttributes);

	//initialize the condition so it can be used across processes
	pthread_condattr_t conditionAttributes;
	pthread_condattr_init(&conditionAttributes);
	pthread_condattr_setpshared(&conditionAttributes,
					PTHREAD_PROCESS_SHARED);
	pthread_cond_init(&dataPtr->condition,&conditionAttributes);

	if((firstChildPID = fork()) == 0) {
		for(int i=0;i<20;i++) {
			pthread_mutex_lock(&dataPtr->mutex);
			dataPtr->data = i * 2;
```

```
                cout << "First child just wrote " << dataPtr->data << endl;
                pthread_mutex_unlock(&dataPtr->mutex);
                pthread_cond_signal(&dataPtr->condition);
                sleep(1);
        }
        exit(0);
    }
    if((secondChildPID = fork()) == 0) {
        for(int i=0;i<20;i++) {
                pthread_mutex_lock(&dataPtr->mutex);
                pthread_cond_wait(&dataPtr->condition,&dataPtr->mutex);
                cout << "Second child just read " << dataPtr->data << endl;
                pthread_mutex_unlock(&dataPtr->mutex);
        }
        exit(0);
    }
    waitpid(firstChildPID,&exitStatusOne,0);
    waitpid(secondChildPID,&exitStatusTwo,0);
}
```


Appendix Four

Windows Concurrency Basics

garth@ggilmour.com

Intro to Windows Concurrency

- Concurrency on windows is a large and complex topic
  - Especially as the same entities can be manipulated through two distinct but very similar API's
  - The native C++ API and the managed .NET libraries
- We need to become familiar with the core concepts
  - Kernel objects and handles
  - Processes, threads and thread pools
  - Thread priorities and affinity to CPU's
  - The Windows thread scheduler
  - Options for synchronizing threads

Kernel Objects

- Kernel objects represent resources
  - They are created by the OS on your behalf
  - Examples include events, files, memory-mapped files, jobs, pipes, processes, threads, semaphores and timers
- Kernel objects are entirely opaque to your code
  - Each object uses an unpublished data structure and lives at a location that cannot be reached from your code
- You work with kernel objects in two ways
  - Windows provides functions to create and modify them
  - You are provided with handles to link you to the objects

Using Handles to Kernel Objects

- A handle is a process relative ID
  - Either 32 or 64 bit depending on the version of Windows
- The handle value is simply an index into a table
  - Each process has its own table of handles in use
  - How the table works is not formally documented
- Kernel objects are released via 'CloseHandle'
  - This does necessarily destroy the object
    - As any number of other processes might also be using it
  - Instead a usage count is decremented till it hits zero
  - When the process ends any open handles are closed

Kernel Objects Across Processes

- A kernel object can be used in many processes
  - But using the handle ID across processes will not work
  - This is to prevent a faulty or malicious application crashing another program by accessing its kernel objects
- There are 4 strategies for sharing kernel objects:
  - When process 'A' spawns 'B' handles can be shared
    - This is known as 'object handle inheritance'
    - It needs to be explicitly enabled in 'CreateProcess'
  - Many kernel objects can be given a machine wide name
    - From Vista onwards private namespaces are also available
  - The 'DuplicateHandle' function is available
    - This copies entries from one handle table to another

Processes in Windows

- In Windows the term 'process' can refer to
  - An address space that contains:
    - All the code and data for an executable or DLL
    - Dynamically allocated memory for heaps and stacks
  - The kernel object used by the operating system to manage and store data about this address space
  - Any handle(s) that we may have to this object
- A process is described as an 'inert entity'
  - To do anything it must have at least one thread running
  - A 'primary thread' will be created and started by the OS

Processes in Windows

- The primary thread enters your code via:
  - The normal 'main' function for console apps
    - '_tmain' if you are using Unicode strings
  - The 'WinMain' function for GUI based apps
    - Similarly '_tWinMain' if you are using Unicode
- The linker embeds startup code in your binary
  - Based on which type of application you choose to build
    - The switch is '/SUBSYSTEM' with two possible options
      - The options are 'CONSOLE' and 'WINDOWS'
  - This code initializes the C/C++ libraries, creates global variables, retrieves command line arguments etc...
    - If the application is console based a text window is created

Processes in Windows

- A process may contain many modules
  - Usually one for the EXE and one for each DLL
  - It is possible to get the name and starting address of each
    - Usually in order to load resources such as images
- A process normally exits when the entry point function of the primary thread (e.g. 'WinMain') returns
  - This triggers cleanup code that frees up resources
- There are other options for ending the process
  - Any thread can call the 'ExitProcess' function
  - A thread in another process can call 'TerminateProcess'
  - We will return to this in more detail later...

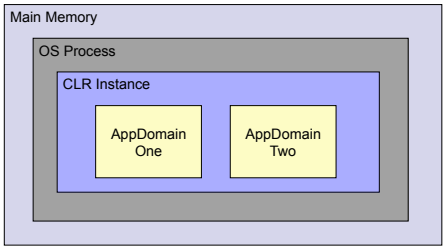
Sharing Data Between Processes

- You app may create multiple processes
  - Especially when you want your main process to be insulated from the code you are about to run
  - The parent can either wait for the child to terminate or release all handles and allow it to run detached
- The child will require some of the parents state
  - And possibly also need a way to pass results back
- This is achieved via Inter-Process Communication
  - As in UNIX there are a range of IPC mechanisms
- Memory mapped files are the most common option
  - They make a regular file part of the process address space

Processes Verses AppDomains

- The Common Language Runtime is the Microsoft VM
  - It can be hosted within an arbitrary process
- All managed programs run within the CLR
  - Which in turn is sub-divided into Application Domains
- An AppDomain resembles an OS Process
  - It is the CLR address space reserved for a program
- Every object is local to an AppDomain
  - You can only reference to objects in the same domain
  - Types are loaded on an per AppDomain basis
    - Static fields can hold different values in different domains

Application Domains



Threads in Windows

- A thread is essentially a kernel object and a stack
  - Note that when running its state also includes values in memory buffers (e.g. registers) on a processor
  - This state is commonly referred to as its context
- There are actually two stacks per thread
  - One for user operations and one for kernel operations
- Threads in Windows are much lighter than processes
  - Fewer system resources are needed to manage them
  - All their memory comes from the process address space
- The only thread created for you is the primary thread
  - You can create others up to a predefined (very large) limit

Native Verses Managed Threads

- Since .NET there have been 2 types of thread:
  - Native threads are those created in C/C++ code
  - Managed threads are those created in C# etc...
- Most of the concepts remain exactly the same
  - The CLR maps managed threads to native ones
    - Extra information needs to be stored (on a thread local basis) in order to implement services such as garbage collection
  - Most C/C++ functions have their .NET equivalents
    - E.g. for starting/stopping threads and setting priority levels
- There is one very important distinction
  - The hosting process can redefine the threading strategy

Thread Scheduling on Windows

- Scheduling is preemptive and priority based
  - A thread is only allowed a set period of time to run
    - This period is commonly known as the quantum
  - Threads are assigned to cores by their priority
- The value of the quantum varies between OS
  - 2 clock intervals for client OS's and 12 for server ones
  - This reflects the fact that client apps are usually concerned with responsiveness and server ones with throughput
- Process boundaries do not influence scheduling
  - If 9 threads from 3 processes are being scheduled it does not matter if the split is 3/3/3, 2/2/5, 7/1/1 etc...

Thread Scheduling on Windows

- When a quantum expires a context switch occurs
  - The current state of the thread is stored into a data structure called a context so another can be scheduled
- It is not guaranteed that the thread will be evicted
  - If it has a higher priority than other threads or there are spare cores then it will likely be given another quantum
- A context switch also occurs when a thread blocks
  - This can happen explicitly from calls in the code (e.g. I/O operations) or implicitly (such as virtual memory paging)
  - If a thread blocks voluntarily it retains the value in its quantum, so it may not run for long when rescheduled

Scheduling and Thread States

Thread State	Description
Initialized	The thread is being created by the OS
Ready	The thread is good to go and is in a queue waiting to be allocated to a processor by the scheduler
Running	The thread is actively running on a processor
Standby	The thread has been selected to run but is not yet executing on a processor
Terminated	The thread has finished execution and will be destroyed when all handles to it are released
Waiting	The thread is not available for scheduling (blocked)
Transition	The thread is unavailable to run because it is waiting for some of its state to be paged back into main memory

Scheduling and Priority Levels

- Every thread has a priority level
  - These can range from 1 to 31 inclusive
- The priority level is made up of two parts
  - These are the priority class and the relative priority
  - There are 6 priority classes and 7 relative priorities
    - Each priority class has a range and a default value
    - The relative priority adds an offset
- It is possible to programmatically adjust priorities
  - This is discouraged as it can interfere with system tasks
  - Device drivers and parts of the kernel run with high priority

Scheduling and Priority Levels

Priority Classes

Title	Range	Default
Real Time	16-31	24
High	11-15	13
Above Normal	8-12	10
Normal	6-10	8
Below Normal	4-8	6
Idle	1-6	4

Relative Values

Title	Modifier
Time Critical	Lifts 'Real Time' to 31 and anything else to 15
Highest	+2
Above Normal	+1
Normal	+0
Below Normal	-1
Lowest	-2
Idle	Drops 'Real Time' to 15 and others to 1

Adjustments in Scheduling

- In abstract scheduling is very simple
  - Visualize the threads as being grouped into queues according to priority level, with the scheduler trying to empty the queues in strict order of priority
- In reality it cannot be that straightforward
  - There would be 'priority inversions' when a low priority thread held a resource required by a high priority one
  - Low priority threads doing I/O might experience timeouts
- The scheduler periodically makes adjustments
  - It will allocate extra time to a thread or boost it to a higher priority based on a range of special circumstances

Adjustments in Scheduling

- A threads priority is only boosted temporarily
  - The priority level will drop one point per quantum until it returns to the level it was at prior to being boosted
- Boosting will occur to a thread when:
  - The scheduler detects it has been starved of CPU time
  - An event or lock becomes available
  - A message is posted to a GUI thread
  - An I/O operation completes
  - A window is in the foreground
  - In Vista when multimedia is played

Scheduling and Thread Affinity

- The scheduler matches a thread to its 'ideal processor'
  - For each thread a processor is selected, based on an algorithm that tries to distribute the load across the cores
  - If the ideal processor is unavailable it prefers to allocate a thread to the processor on which it most recently executed
    - This is the most efficient way of managing the hardware
- Threads and processes can be assigned a processor
  - This is known as 'CPU Affinity' and usually discouraged
- Affinity given to a process is inherited by all its threads
  - Whereas assigning an affinity to an individual thread has no effect on the allocation of other threads in the process

Thread Pooling on Windows

- So far we have assumed explicit thread creation
  - Where you manually create a new thread for each task
- This approach is often inefficient
  - Particularly when you have a large number of quick-running tasks that you want to run in parallel
- A better solution is a pool of worker threads
  - Each thread in the pool waits until assigned a task
  - This avoids the overhead of startup and shutdown
- Windows has extensive support for thread pooling
  - Both at the managed and the native levels
  - Pooling is recommended for most programs

Thread Pooling on Windows

- There are three Windows thread pools:
  - One introduced in Windows 2000
    - And still available for backwards compatibility
  - One introduced in Windows Vista
    - Which completely replaces the earlier version
  - One available only within the CLR
- The Vista pool allows you to submit a function so that:
  - It can be run on a worker thread
  - It can be called at intervals via a timer
  - It will run when a kernel object is signalled
  - It will run when an I/O operation completes

Windows Threads and Fibres

- Windows thread scheduling is performed by the kernel
  - No scheduling activity is performed in user mode
- Other operating systems take a mixed approach
  - The kernel allocates each process a quantum
  - The process them divides that time amongst its threads
  - This is known as 'green threads' on Solaris UNIX
- Scheduling threads in user-mode has two advantages
  - There are fewer context switches for the OS to manage
  - It is easier to cope with situations where multiple threads need to work closely together and share data frequently

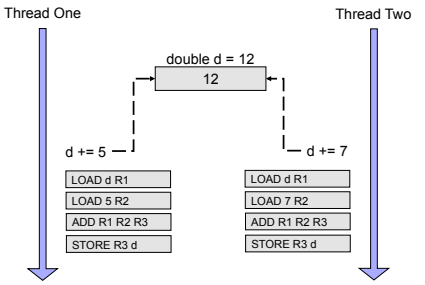
Windows Threads and Fibers

- Windows supports 'lightweight threads' via fibres
  - This support is only available in native code
- A single thread can contain many fibres
  - The kernel is unaware of the fibres existence
  - In the code you first convert a normal thread into a fibre and then create as many others as you need
- Scheduling of fibers is co-operative
  - The currently running fiber keeps executing until it voluntary yields the CPU by calling 'SwitchToFiber'
    - Note that this has no effect on normal thread scheduling
  - There is support available for Fiber Local Storage (FLS)

Synchronizing Access to Data

- Data shared between threads is a source of errors
  - If the machine instructions for accessing data overlap intermittently then your code will contain 'heisenbugs'
  - The lower the frequency of data sharing the better
- Hardware can influence if an error actually occurs
  - The width of registers, the size of on-chip caches and the strategies for buffering values from RAM can all play a part
  - The number of cores, the overall workload and the scheduling policy are obviously factors as well
- Data shared between threads needs to be protected
  - There are many different ways of achieving this

Q: What Value Will End Up In ‘d’



Synchronizing Access to Data

- Both native and managed code provide mechanisms for preventing race conditions
  - Here we will be concerned only with native code
- There are two different types of mechanism
  - Those that operate almost entirely in user mode
    - These are fast put only work within processes
  - Those that reply on kernel objects
    - These work across processes and are very flexible
    - But they are slow as your code must switch into kernel mode
- Note there is no way of erecting a ‘magic shield’
  - No special keyword you can put on a variable declaration

Synchronization Mechanisms

User Mode

- Atomic operations
- Critical sections
- Slim reader-writer locks
- Conditional variables

Kernel Mode

- Files and streams
- Events
- Timers
- Semaphores
- Mutexes

Using Atomic Operations

- The first type of user-mode synchronization is the 'InterlockedXXX' family of functions
  - These enable you to modify the value of a variable or pointer in a single atomic operation
  - Whilst the value is being loaded onto a register, changed and stored back it cannot be accessed by another thread
- A simple example is 'InterlockedIncrement'
  - This adds atomically adds one to the value of a variable
- How the functions work depends on the hardware
  - Some chips now have special atomic operations

Using Critical Sections

- If you have multiple lines of code that access shared data you should use a critical section
  - A critical section is an opaque data structure
  - You are free to create as many instances as you like
- Only one thread can be within a critical section
  - Functions are provided to enter and leave the section
- If the section is held by then the calling thread blocks
  - In practise it spins a number of times before waiting
    - This is because critical sections are usually short, whereas waiting involves transitioning into and out of kernel mode
  - It is also possible to test without blocking

Using Slim Reader-Writer Locks

- Sometimes single-threaded access to data is overkill
  - Instead you want to allow any number of threads to traverse a data structure, but only one to change it
- This is achieved via a 'Slim Read-Write Lock'
  - As with a critical section a SRWLock is an opaque data structure that you create and use via functions
  - But you request the lock in shared or exclusive mode
- The SRWLock does have some disadvantages:
  - There is no way to acquire the lock without risking a wait
  - The lock cannot be acquired recursively

Using Conditional Variables

- Conditional variables implement a common pattern
  - Where you want a thread to acquire a lock and then
    - **Either** read some data and release the lock
    - **Or** release the lock and start waiting on an event
      - So that you can be notified when the data becomes available
  - There is a nasty race condition within the pattern
    - The notification event might fire **between** your thread releasing the lock and beginning to wait, leading to an infinite wait
- Conditional variables make the release-to-wait atomic
  - You create the variable and then call a blocking function
    - Passing in the variable and a lock as parameters
  - The lock can either be a critical section or a SRWLock

Synchronizing with Kernel Objects

- Some kernel objects support signalling
  - They can be in either a signalled or unsignalled state
    - The details vary according to the type of the object
    - E.g. a process or thread object is initialized to unsignalled on creation but switches state when it terminates
  - The types of object are processes, threads, jobs, files, streams, events, timers semaphores and mutexes
- A thread can wait for such a kernel object
  - The 'WaitForSingleObject' function call is one of the most commonly used in Windows programming
    - Some objects reset their state after a successful wait
      - This means that if multiple threads are waiting only one wakes

Synchronizing with Kernel Objects

- An event is the simplest type of kernel object
  - Its only purpose is to signal that something has occurred
  - An event can be manual-reset or auto-reset
    - When a manual-reset object is signalled any and all waiting threads become schedulable
    - Whereas with an auto-reset event only one is woken
- A timer is a kernel object that signals itself
  - Either at a certain time or at regular intervals
  - Once again it can be either manual or auto reset
  - You can optionally provide a function pointer
    - Which you want called asynchronously when the timer fires

Synchronizing with Kernel Objects

- A semaphore is used for resource counting
  - It contains a minimum and maximum count
  - Client code can increment or decrement the counts
    - But never to a value below the minimum or above the maximum
  - The semaphore is unsignalled when the resource count is at the minimum and signalled otherwise
- A mutex is similar to a critical section
  - It lets you ensure exclusive access to a single resource
  - The mutex is unsignalled when it is not owned
    - By waiting on a mutex you are trying to acquire ownership
  - There is built in support for recursion

Working With Modules

- A Process can contain many modules:
  - The executable containing the start function
  - Any number of libraries (DLL's)
- The divisions between modules remain visible
  - You may need this information for loading resources

Function	Description and Major Parameters
GetModuleFileName	Retrieves the path to the file from which a module was loaded 1. A handle to the module NB if NULL the EXE of the current process is assumed 2. The address of a string inside which the path will be placed 3. The length of the string
GetModuleFileNameEx	As above but can be called from another process
GetModuleHandle	Returns the address where a module resides in the process 1. A string containing the name of an executable or DLL

Creating and Destroying Processes

Function	Description and Major Parameters
CreateProcess	Creates a new process and its primary thread 1. The name of the executable file 2. The command line string 3. Security attributes for the process 4. Security attributes for the thread 5. A flag for handle inheritance 6. A set of miscellaneous flags 7. A list of environment strings 8. The processes working directory 9. Startup info for the program window 10. A structure that will be populated with id's
ExitProcess	Terminates the process 1. The exit code of the process
TerminateProcess	Asynchronously terminates the process. Can be called by a thread in the current process or a different one. 1. The handle of the process to terminate 2. The exit code of the process

Creating and Destroying Threads

Function	Description and Major Parameters
CreateThread	Allocates memory for and initiates a new thread 1. Security attributes for the new thread 2. The space to be made available for the threads stack 3. The address of the function to act as the 'main' 4. Arbitrary information to be passed to the thread 5. A set of miscellaneous flags 6. A DWORD in which the ID of the new thread is stored
_beginthreadex	Performs thread specific initialization of the C/C++ runtime libraries and then calls 'CreateThread'
ExitThread	Ends the current thread 1. The exit code of the thread
_endthreadex	Performs C/C++ specific tidy-up and then calls 'ExitThread'
SetErrorMode	A bitmask of flags specifying how the process should respond to unhandled exceptions and other failures 1. The bitmask as an unsigned integer

Creating and Destroying Processes

- Calls to 'ExitXXX' stop a function in mid-execution
  - Any code placed after the call will never run
- Running to completion is the best way to end a thread
  - 'ExitProcess' is called after 'main' completes
- The primary thread is able to call 'ExitThread'
  - But even if 'main' makes the call the process will remain as long as one or more programmer-created threads is active
  - However because the primary thread does not run to completion the C/C++ runtime will not perform clean-up
    - Leading (for example) to destructors not being called

Determining Your Identity

- Windows uses special values to represent the current process and the currently running thread
  - These are distinct from the actual handles
- Sometimes you need the actual handle
  - Such as when you pass one thread to another

Function	Description and Major Parameters
GetCurrentProcess	Returns a 'magic number' that represents the current process. It can be turned into a real handle by calling 'DuplicateHandle'
GetCurrentThread	Same as above, but the number represents the current thread
GetCurrentProcessId	Returns the id of the kernel object representing the process
GetCurrentThreadId	Returns the id of the kernel object representing the thread

Performing Atomic Operations

Function	Description and Major Parameters
InterlockedIncrement InterlockedIncrement64	Atomically increments the value at the specified address
InterlockedDecrement InterlockedDecrement64	Atomically decrements the value at the specified address
InterlockedExchange InterlockedExchange64	Incrementally sets a variable to a value 1. The address of the variable to set 2. The value to set the variable to
InterlockedExchangeAdd InterlockedExchangeAdd64	As above but adds to the existing value
InterlockedExchangePointer	Sets a pointer to the specific value

Using Critical Sections

Function	Description and Major Parameters
InitializeCriticalSection	Initializes a CRITCAL_SECTION data structure. Required because the structure has an opaque format. Must be called before 'EnterCriticalSection' is called
EnterCriticalSection	Marks the current thread as owning the critical section, or else places the thread in a wait state via a kernel event. Note that the section is re-entrant (can be acquired multiple times)
TryEnterCriticalSection	As above but returns a boolean value and does not place the calling thread into a waiting state
LeaveCriticalSection	Decrements the count of the number of times the thread has acquired the resource. If the count is zero then the section is marked as available for another thread to acquire
DeleteCriticalSection	Un-sets the fields of the critical section

Using 'SRWLock'

Function	Description and Major Parameters
InitializeSRWLock	Initializes an SRWLOCK data structure 1. The address of the lock
AcquireSRWLockExclusive	Used by a writing thread to acquire the lock 1. The address of the lock
ReleaseSRWLockExclusive	Used by a writing thread to release the lock 1. The address of the lock
AcquireSRWLockShared	Used by a reading thread to acquire the lock 1. The address of the lock
ReleaseSRWLockShared	Used by a reading thread to release the lock 1. The address of the lock

Working with Kernel Objects

Function	Description and Major Parameters
WaitForSingleObject	One of the most important API functions. Causes a thread to enter a waiting state until a kernel object becomes signalled. 1. A handle to the kernel object you want to wait on 2. How long the thread will wait for the object
WaitForMultipleObjects	As above but for between 1 and 64 objects 1. The number of handles you want to wait on 2. The address of an array of handles 3. A boolean value indicating if you want to wait till all the objects become signalled. Otherwise the first one wakes the thread 4. How long the threads will wait for the objects
CreateWaitableTimer	Builds a timer in either manual-reset or auto-reset mode
OpenWaitableTimer	Lets a thread acquire its own handle to an existing timer
SetWaitableTimer	Activates a timer so it will fire after a set time
CancelWaitiableTimer	Cancels a timer so it will never go off

Working with Kernel Objects

Function	Description and Major Parameters
CreateSemaphore CreateSemahoreEx	Creates a semaphore with initial and maximum counts
OpenSemaphore	Allows a thread to acquire a handle to an existing semaphore
ReleaseSemaphore	Increments a semaphores release count
CreateMutex CreateMutexEx	Creates a mutex
OpenMutex	Allows a thread to acquire a handle to an existing mutex
ReleaseMutex	Causes a mutex to become unowned and therefore signalled

Working With Fibres

- The API for fibres is straightforward
  - Note that the code which calls 'ConvertThreadToFiber' continues executing but now as a fiber
  - It will keep running until is yields to another

Function	Description and Major Parameters
ConvertThreadToFiber	Allocates memory for the fibres execution context and converts the running code into a fiber 1. Arbitrary data that can be retrieved with 'GetFiberData'
CreateFiber	Creates a new fiber 1. The size of the user-mode stack to create NB can be zero for the default size 2. The address of the start routine 3. Arbitrary data
SwitchToFiber	Enables the executing code to give runtime to another fiber 1. A pointer to the fiber