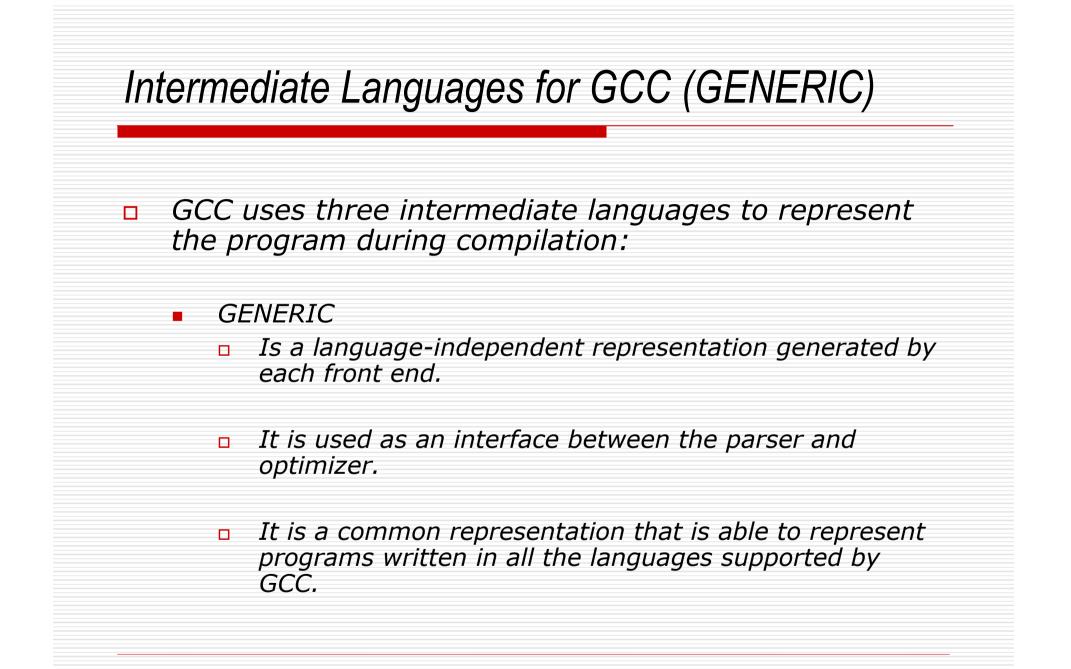
Compiler Theory

(GCC – the GNU Compiler Collection)

Sandro Spina 2009

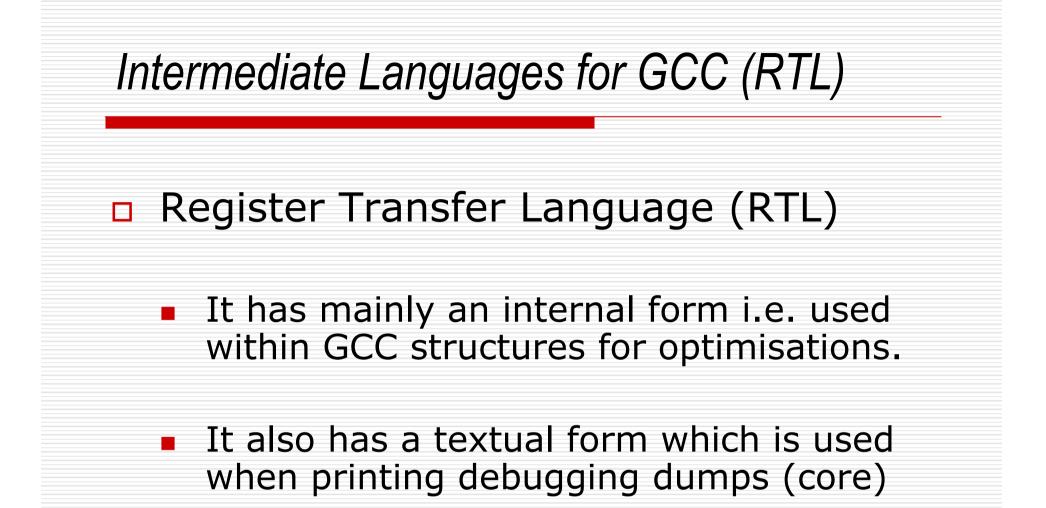
GCC

- Probably the most used compiler.
- Not only a native compiler but it can also cross-compile any program, producing executables for different systems other than the one it is being used on.
- GCC is written in C and can compile itself !!
- Provides numerous front-ends
 - C
 - C++
 - Objective-C
 - Fortran
 - Java
 - Ada



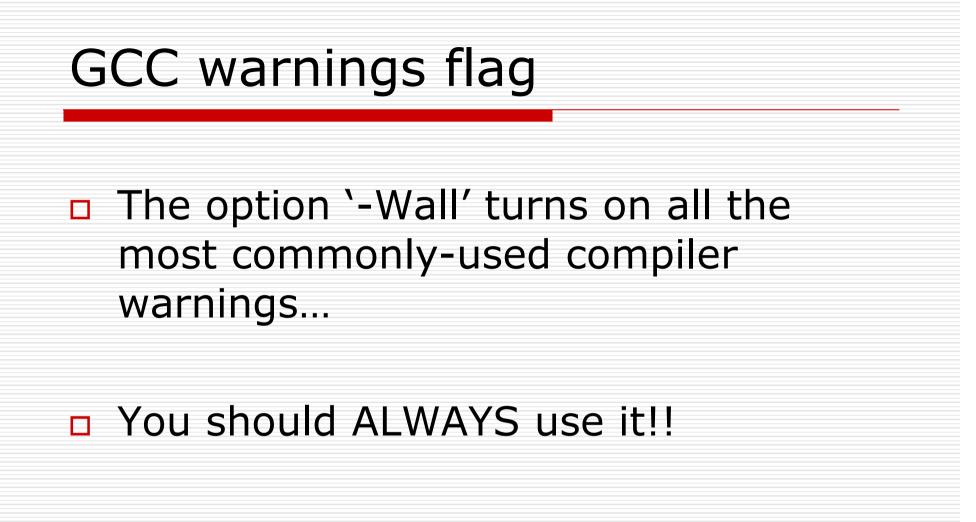
Intermediate Languages for GCC (GIMPLE)

- Gimlipification (done by the gimplifier) is the compiler pass which lowers GENERIC to GIMPLE. It works recursively by replacing complex statements with sequences of simple statements.
 - GIMPLE
 - Used for target and language independent optimisations (e.g. inlining, constant propagation, etc)
 - It is a language independent, tree based representation
 - Differs from GENERIC in that the GIMPLE grammar is more restrictive: expressions contain no more that 3 operands (except for functions calls)
 - It has no control flow structures because these are lowered to gotos



Compiling a C Program

- Compilation refers (as you all obviously know by now) to the process of converting a program from source code (text), in a programming language such as C or C++, into machine code, i.e. a sequence of 1's and 0's used to control what the processing unit (CPU, GPU, etc.) does.
- \$ gcc -Wall hello.c -o hello
- The above compiles the source code in `hello.c' to machine code and stores it in the executable file `hello'
- -o specifies the output file



 Because they are essential when detecting problems and debugging.

GCC warnings example



{

}

printf ("Three plus two is %f\n", 5); return 0;

- \$ gcc -Wall example.c -0 example
 Example.c: In function `main':
 Example.c:6: warning: double format, different type arg (arg 2)
- Warnings do not prevent compilation ... but indicate possible problems.
- The program above compiles (and runs) but gives an incorrect answer.

Compiling multiple source files

- Large programs will always be split in multiple files (if your's are not then you might consider it)
- The nice things about this is that one can compile the individual parts independently.
- Header files are used to specify the prototype of function calls.
 Eg. void hello (const char * name);

```
#include ``hello.h"
int main (void)
    {
        hello (``world");
        return 0;
        }
        hello (``world");
        return 0;
        }
    }
        hello (``world");
        return 0;
        }
        hello (``world");
        return 0;
        }
    }
```

Compiling multiple source files

- \$ gcc –Wall main.c hello_fn.c –o hello
- Note that the header file is not included in the list of files to be compiled.
- This is because the directive #include "hello.h" in the source files instructs the compiler to include it automatically at the appropriate points.
- This is something which is carried out by the pre-processor tool cpp.
- Only the files which have changed need recompilation because a twostage process is carried out ... compilation and linking

Compiling then linking ...

- In the first stage the source file is compiled without creating an executable.
- The result is referred to as an *object file*, and has the extension .o when using gcc compiler.
- In the second stage, the object files are merged together by a separate program called a *linker*.
- The linker combines all the object files together to create an executable.
- Essentially, an object file contains machine code where any references to the memory addresses of functions (or variables) in other files are left undefined.
- This allows source files to be compiled without direct reference to each other.
- The linker fills in these missing addresses when it produces the executable.

Creating object files from source files

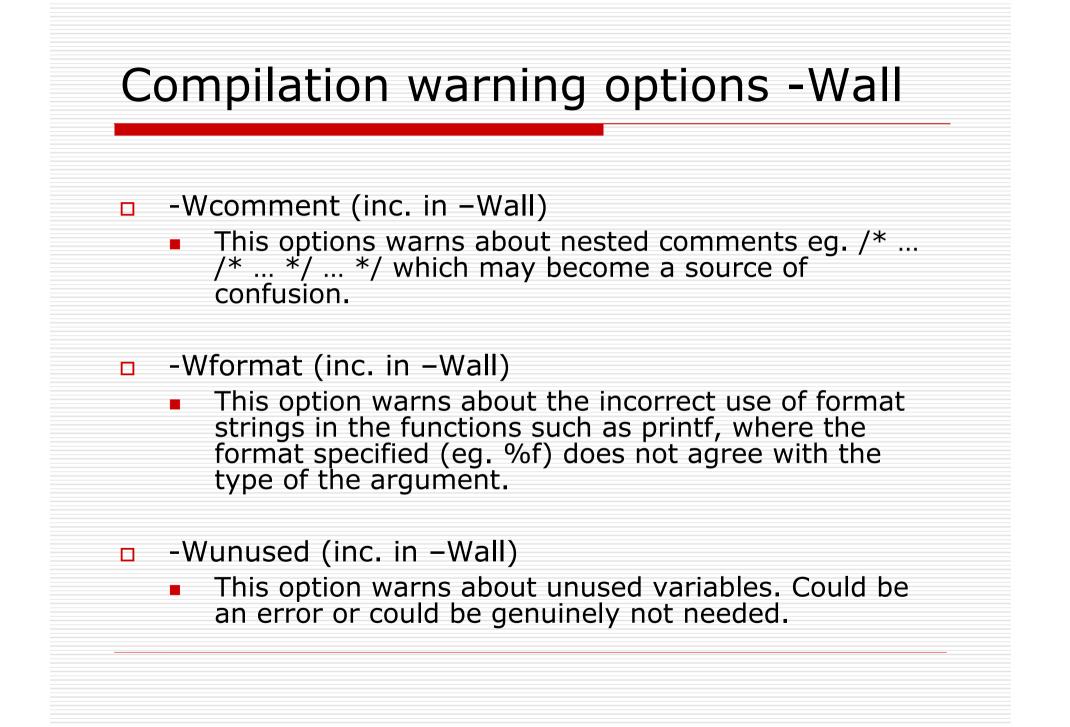
- □ \$ gcc –Wall –c main.c
- Produces an object file `main.o' containing the machine code for the main function with a reference to the external function hello.
- The corresponding memory address is left undefined at this stage
- \$ gcc main.o hello_fn.o -o hello
- **gcc** uses the GNU linker *ld*
- It should be noted that linking is effectively an unambiguous process which either succeeds or fails (and it fails only if there are references which cannot be resolved)
- Warnings flag is useless here of course!

Linking with external libraries

- A library is a collection of precompiled object files which can be linked into programs. Eg. Math library *libm.a*
- Static libraries: They are created from object files with GNU archiver tool ar, and are used by the linker to resolve references to functions at compile-time.
- □ \$ gcc –Wall add.c –Im –o add
- -I is used to link against libraries. –Im for math one.
- Shared libraries: libraries are loaded and references are resolved at run-time.

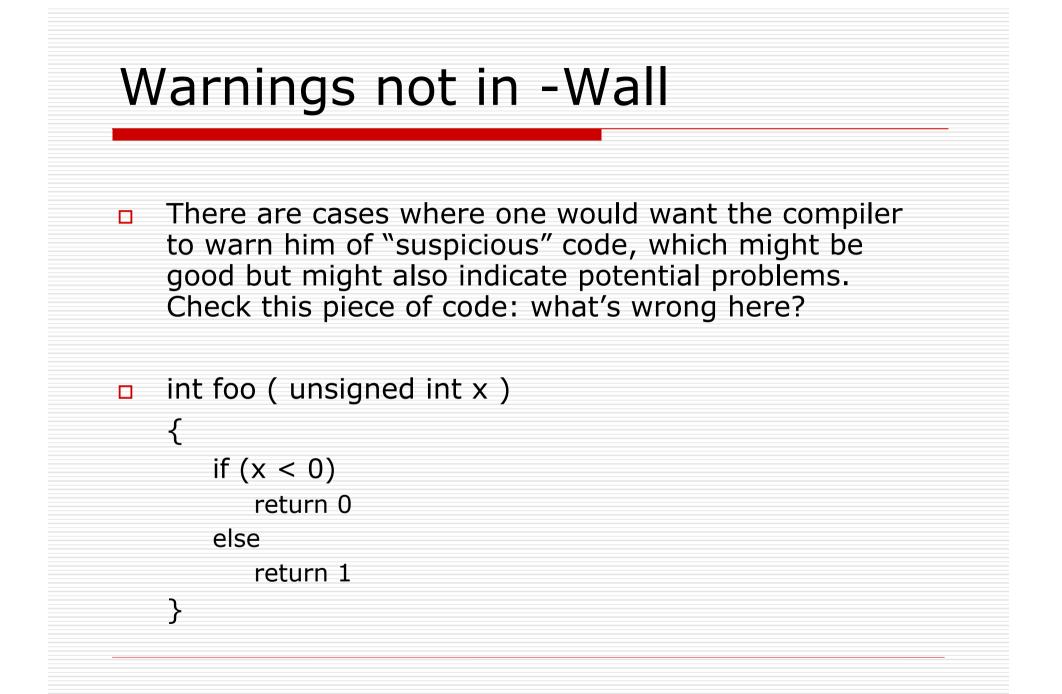
Linking with shared libraries

- When a program is linked against a static library, the machine code from any external functions used by the program is copied from the library into the final executable (increasing it's size)
- With shared libraries a more advanced from of linking is performed, which makes the executables smaller.
- Shared library ext. is `.so' for shared objects
- Instead of complete machine code of functions the executable would contain a small table of the functions it requires.
- Before executable starts running, the machine code of the external functions is copied into memory from the shared library dynamic linking
- Dynamic linking makes executables smaller.



Compilation warning options -Wall

- -Wimplicit (inc. in –Wall)
 - This options warns about any functions which are used without being declared. Usually you're missing the #include header file.
- -Wreturn-type (inc. in –Wall)
 - This option warns about functions which are declared without a return type but not declared *void*. It also catches empty return statements in functions that are not declared *void*. It is usually good to avoid ambiguity (eg. Use `return 0' not just `return').
- Any compiler warning can be taken as an indication of a potentially serious problem. Good compiler implementations try an pinpoints these cases.



-W gcc compiler option

- The '-W' is a general option which warns about a selection of common programming errors, such as functions which can return without a value and comparisons between signed and unsigned values.
- From the previous example the compiler outputs:
- \$ gcc -W -c example.c
 Example.c: In function `foo'
 Example.c:4: warning: comparison of unsigned expressions < 0 is always false

-W specific options

-Wconversion

 This option warns about implicit type conversions that could cause unexpected results. For eg. unsigned int x = -1;

-Wshadow

 The option warns about the redeclaration of a variable name in a scope where it has already been declared. This is referred to as variable shadowing. Check code in next slide.

-Wshadow

```
    double test ( double x )
```

```
{
    double y = 1.0;
    {
        double y;
        y = x;
    }
    return y;
}
```

This is clearly a valid piece of code but some people might think that the return value of y = x when it's 1.

Preprocessor - cpp

- It is automatically called whenever gcc processes a c or c++ file. Recently it can been inbuilt in the compiler itself. Cpp still exists.
- Used mainly to expand macros.
- For eg ... <u>#ifdef:</u>
 - #ifdef TEST
 - Printf("Test Mode ... \n");
 - #endif
- The gcc option `-DNAME' defines a preprocessor macro NAME from the command line. In the program above if we want to compile in test mode, the command line option `-DTEST' is used.

Macros with values

- In addition to being defined, a macro can also be assigned a value. Clearly this value is inserted in the source code at each point where the macro occurs.
- -DNAME=value ... eg -DNUM=100 would replace the occurrences of macro NUM with 100 in the program
- □ \$gcc -Wall -DNUM=100 test.c
- □ \$gcc -Wall -DNUM="50+50" test.c
- □ The above a equivalent gcc calls.
- Macros can also be defined inside the code using the #define command. Eg
 - #define SIZE 100
 - int table1[SIZE]

Compiling with optimisation

- GCC is an optimised compiler, i.e. it provides a number of options which either increase the speed of an executable or else decrease it's size (or both)
- Source-level optimisation
 - Common subexpression elimination
 - Function inlining (especially important when small functions are continuously invoked). Eg double sq(double x) { return x*x; }
 Consider this function inside a loop!

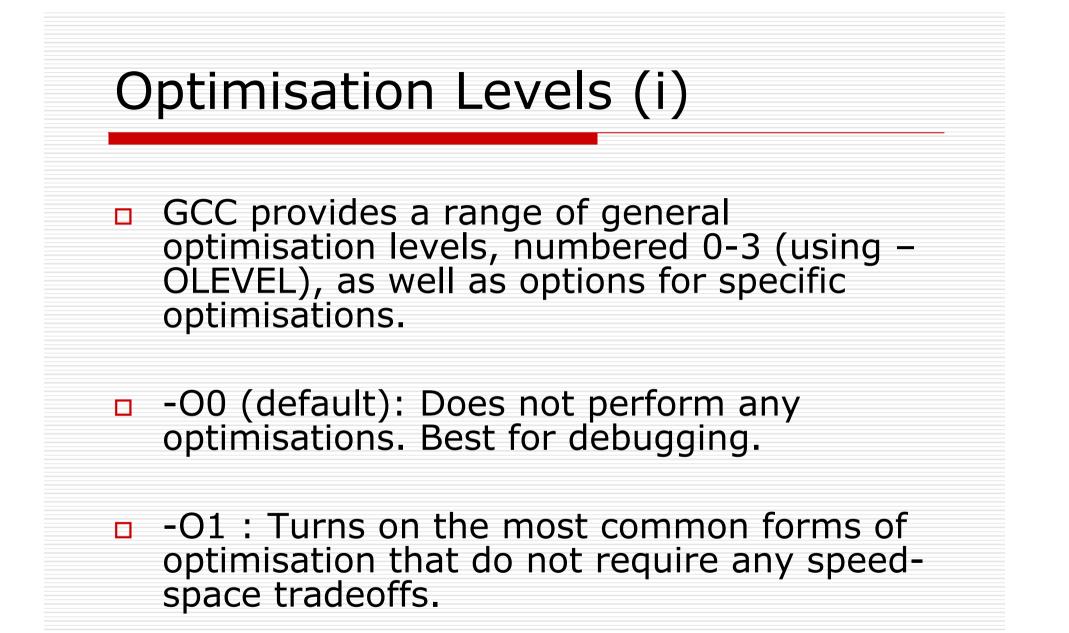
Compiling with optimisation (ii)

Speed-space tradeoffs

- One can produce faster code at the expense of size. Eg. Loop Unrolling
- Increases the speed of loops by eliminating the "end of loop" condition in each iteration. Eg
- For (i=0; i<8; i++) { y[i] = i; }</p>
- Is replaced with ...
- y[0] = 0; y[1] = 1; etc ...

Compiling with optimisation (iii)

- Scheduling:
 - The lowest level of optimisation in which the compiler determines the best ordering of individual instructions.
 - Pipelining, in which multiple instructions execute in parallel on the same CPU.
 - The compiler tries to optimise this amount of parallelisation.
 - Increases speed of executable but takes longer to compiler due to its complexity!





- -O2 : Turns on O1 plus instruction scheduling. It will not increase the executable size but will take longer to compile. Best for release versions.
- -O3 : Turns on speed-space optimisations such as unfolding of loops.
- -funroll-loops : specific optimisation
- -Os: Turns on optimisations which should decrease the size of the executable.

Summary ...

- A single invocation of GCC consists of the following stages:
 - Preprocessing (to expand macros)
 - Compilation (from source code to assembly language)
 - Assembly (from assembly language to machine code)
 - Linking (to create the final executable)

