Numerical Simulation and Scientific Computing I

Lecture 11: Build Systems and Debugging

Xaver Klemenschits, Paul Manstetten, and Josef Weinbub

Institute for Microelectronics
TU Wien

nssc@iue.tuwien.ac.at

WS 2022

Outline

- **Quiz Wrapup**
- Build Systems
- Debugging
- Next Quiz

Quiz Wrapup

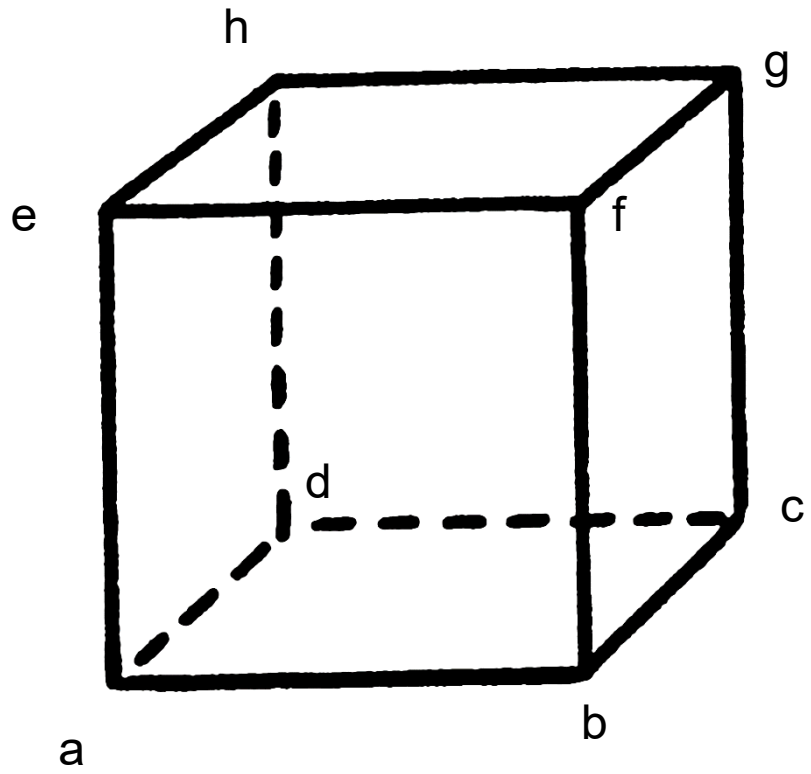
1. What is the Hasse diagram of a cuboid 3-cell?

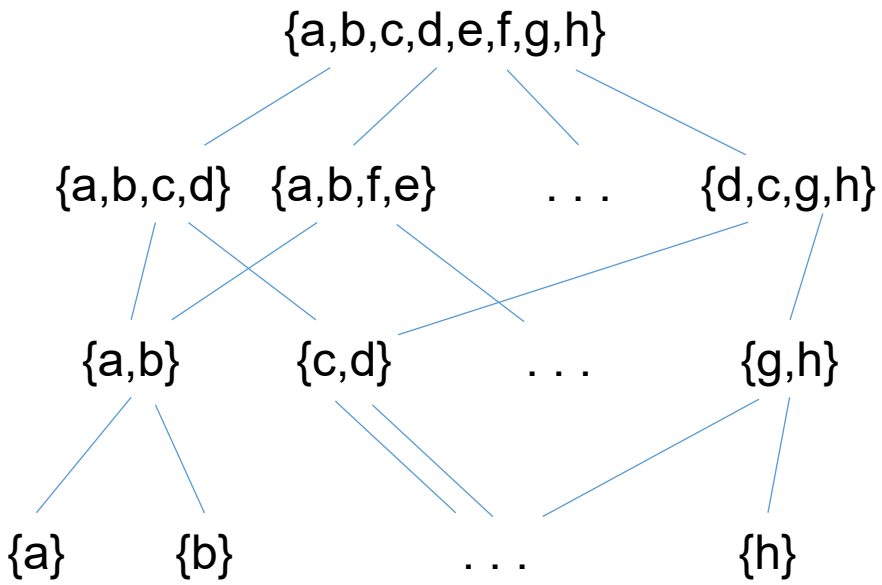
   next slide
2. Is the letter “A” a convex or a concave object?

   concave
3. What is the Genus of the surface of a coffee mug?

   1
4. Do you know a platform independent build system? → Later
5. What is gdb? → Later

What is the Hasse diagram of a cuboid 3-cell?





Outline

- Quiz Wrapup
- **Build Systems**
- Debugging
- Next Quiz

Sources

- Peter Smith
  Software Build Systems: Principles and Experience
  https://www.oreilly.com/library/view/software-build-systems/9780132171953/
  → Accessible via TUW

Motivation

- Anything more than a few source files requires some type of automated build system for efficiency
- Build system:
  Translate human-readable source code into an executable program
- Quickly building workflows gets complicated
- Simplifies software development
- Essential for automated testing
- Rebuild entire software project (including documentation) after changes

Motivation

```
g++ -o sorter main.cpp sort.cpp files.cpp
   tree.cpp merge.cpp
```

- Imagine more source files
- Imagine various compiler flag configurations
  - -g … debug
  - -O1, -O2, -O3 … optimization level 1-3

Solutions

- Shell script?
- Makefile? (Advantage: only rebuilds changed sources)
- Others?

Motivation

Makefile

```
sorter: main.cpp sort.cpp files.cpp tree.cpp merge.cpp
        g++ -o sorter main.cpp sort.cpp files.cpp
          tree.cpp merge.cpp
```

Not efficient

- **Source files listed twice**
- **Forced rebuilds**
- **No header file dependencies**

Motivation

Makefile

```
SOURCES  = main.cpp sort.cpp files.cpp tree.cpp
           merge.cpp

OBJECTS   = $(SOURCES:.cpp=.o)

sorter: $(OBJECTS)
        $(CC) -o $@ $^

$(OBJS): numbers.h
```

We will learn about these parameters later!

Better

- **Break up compilation steps (re-)compiled independently**

Motivation

- Build systems can be very complex and hard to maintain
- Graphical user interfaces help
  - Designed to hide the complexities
    - Pro: Easier to start
    - Con: (1) Potentially important details are hidden
      (2) Handling large software builds with a plethora of dependencies is challenging.
- Build systems often grow with the development
  - Small projects: a single Makefile
  - Eventually, as project grows, typical problem: Source files won't get recompiled or are unnecessarily recompiled (multiple times)
  - Larger projects: a hierarchy of Makefiles (to provide different levels of complexity in different Makefiles levels), or other build systems

The True Cost of a Build System

- **Efficient build system is important for productivity**
- **Average productivity loss 10-30% due to build problems**
  - **Consider the economic impact**

Typical Reasons

- **Bad dependencies: compilation errors (e.g., *missing* dependencies) → forces time intensive *clean builds***
- **Slow compilation: Software projects can take hours to fully compile**

Clean Build:

Remove all previously compiled object files

Kumfert, Gary and Tom Epperly. 2002. *Software in the DOE: The Hidden Overhead of "The Build."* Livermore, CA: Lawrence Livermore National Laboratory

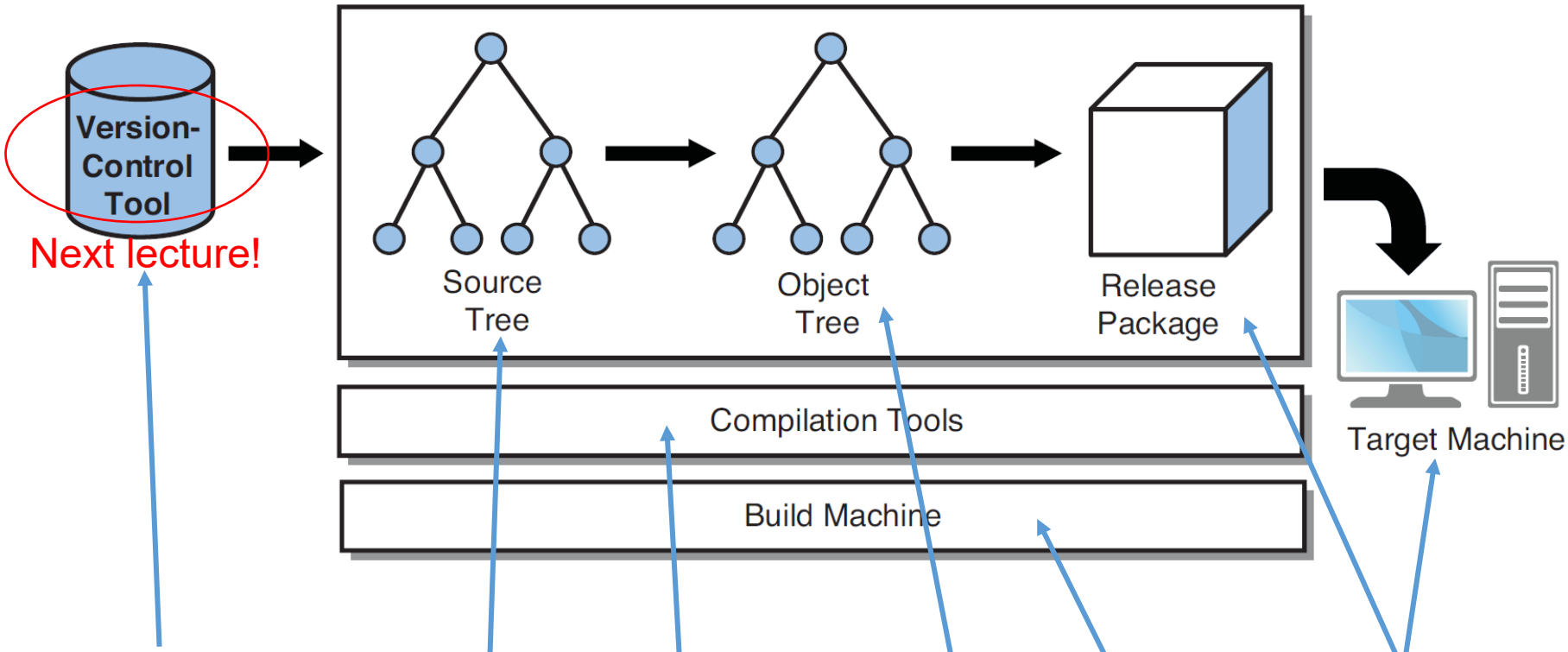
Build Systems for CSE

- **Build systems for all kinds of software projects**
- **Here, focus is on**
  - **Compiled languages, i.e., C++**
  - **CSE typical scenarios**
  - **Non-IDE solutions: You must understand the basic inner workings first.**

The Roles of a Build System

- **Compilation of software written in compiled languages (e.g. C++)**
- **Packaging and testing**
- **Execution of *unit testing***
- **Execution of static analysis tools to identify bugs**
- **Generation of documentations (e.g., HTML, PDF)**

Build System for Compiled Languages



e.g. GIT, SVN
Stores source code; allows shared team development: see next lecture

Checked-out source files; developer local

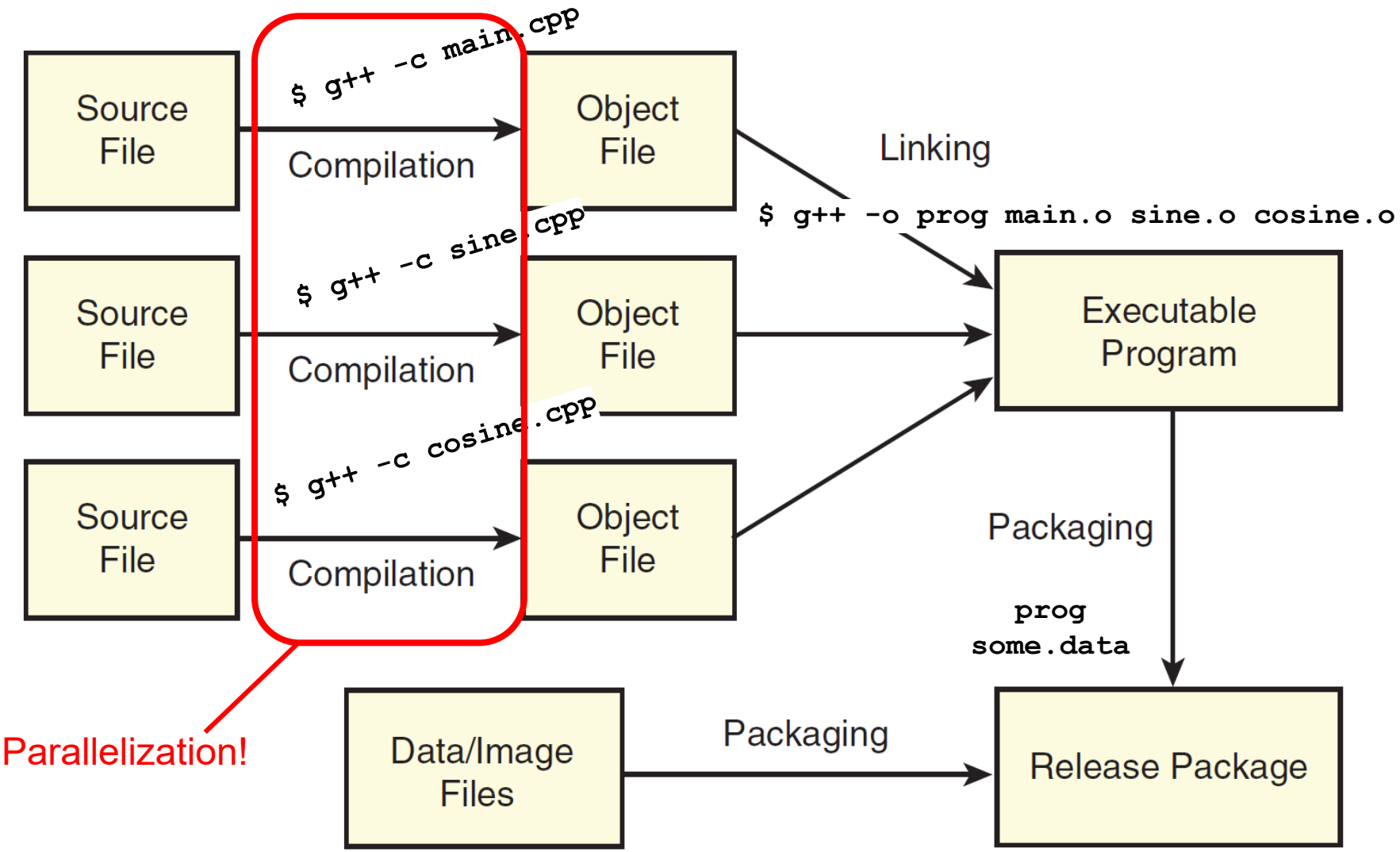
Compiler/Linker, Documentation generator etc

Compiled object files; developer local

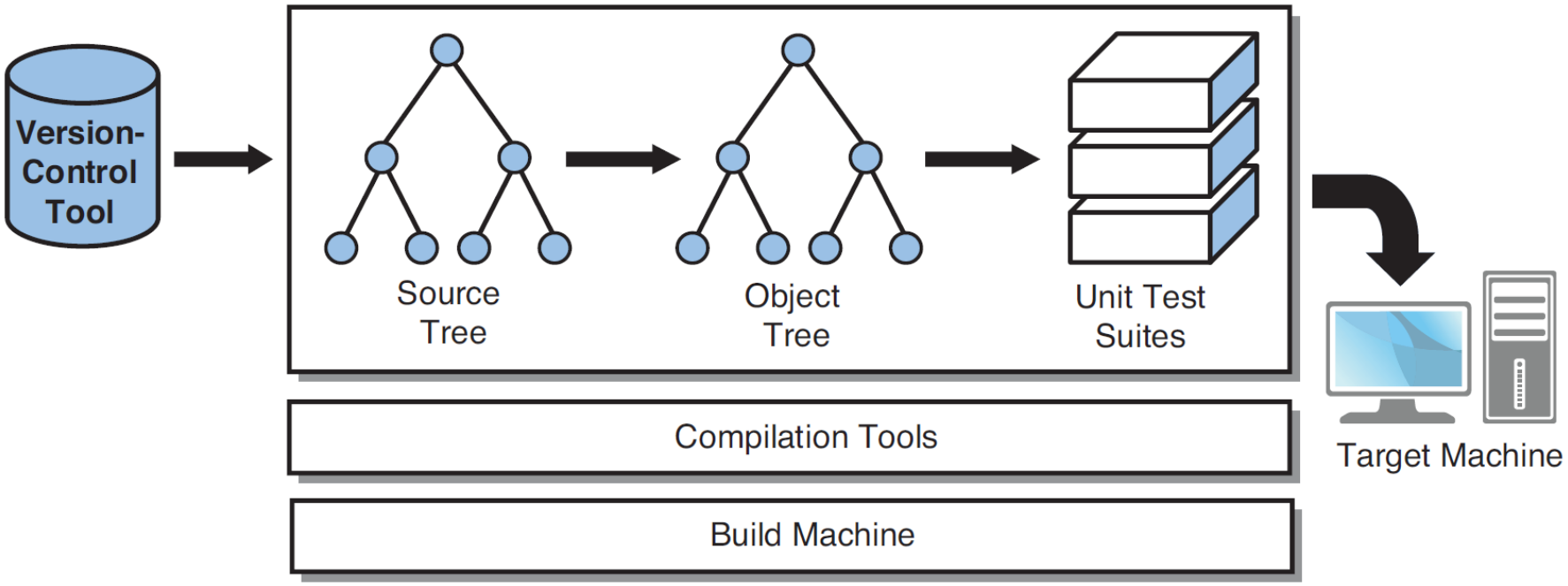
Computers where build system is executed

Method of packaging, distribution, and installation on target machine

Build System for Compiled Languages



Unit Testing



- **Build system produces a set of smaller *unit test suites***
- **Each *suite* is executed on the target machine and produces “pass”/”fail” whether software behaved as expected**
- **E.g. Cpptest** https://github.com/cpptest/cpptest

Unit Testing: CppUnit

Live Showcase!

```
#ifndef BASIC_MATH_HPP__
#define BASIC_MATH_HPP__

class CBasicMath
{
public:
   int Addition(int x, int y);
   int Multiply(int x, int y);
};

#endif
```

```
#include "BasicMath.hpp"

int CBasicMath::Addition(int x, int y)
{
   return (x + y);
}

int CBasicMath::Multiply(int x, int y)
{
   return (x * y);
}
```

Your Library

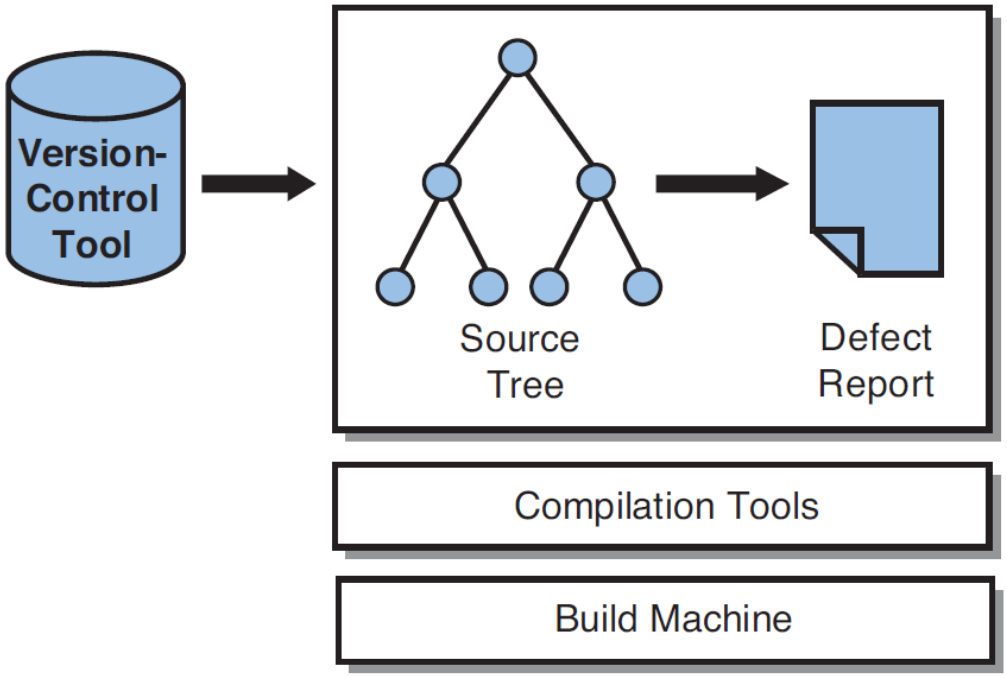
```
class TestBasicMath : public CppUnit::TestFixture {
   CPPUNIT_TEST_SUITE(TestBasicMath);
   CPPUNIT_TEST(testAddition);
   CPPUNIT_TEST(testMultiply);
   CPPUNIT_TEST_SUITE_END();

public:
   void setUp(void)
   {
      mTestObj = new CBasicMath();
   }
   void tearDown(void)
   {
      delete mTestObj;
   }

protected:
   void testAddition(void)
   {
      CPPUNIT_ASSERT(5 == mTestObj->Addition(2,3));
   }
   void testMultiply(void)
   {
      CPPUNIT_ASSERT(6 == mTestObj->Multiply(2,3));
   }

private:
   CBasicMath *mTestObj;
};
```

Your Unit Tests

Unit Testing: CppTest

```
class TestAdd : public Test::Suite
{
    /* ... */
};


int main(int argc, char* argv[])
{

    // Some exemplary setup of unit tests …


    Test::Suite ts;
    ts.add(auto_ptr<Test::Suite>(new TestAdd));
    ts.add(auto_ptr<Test::Suite>(new TestAdd(1,2,3)));
    ts.add(auto_ptr<Test::Suite>(new TestAdd(11,24,35)));

    auto_ptr<Test::Output> output(cmdline(argc, argv));
    ts.run(*output, true);

    return 0;
}
```

Static Analysis



- **Static → build time**
- **Examines source code to find bugs**
- **No object code is compiled; rather a report is generated**
- **E.g. Cppcheck, Coverity Scan**

Static Analysis: Cppcheck

Live Showcase!

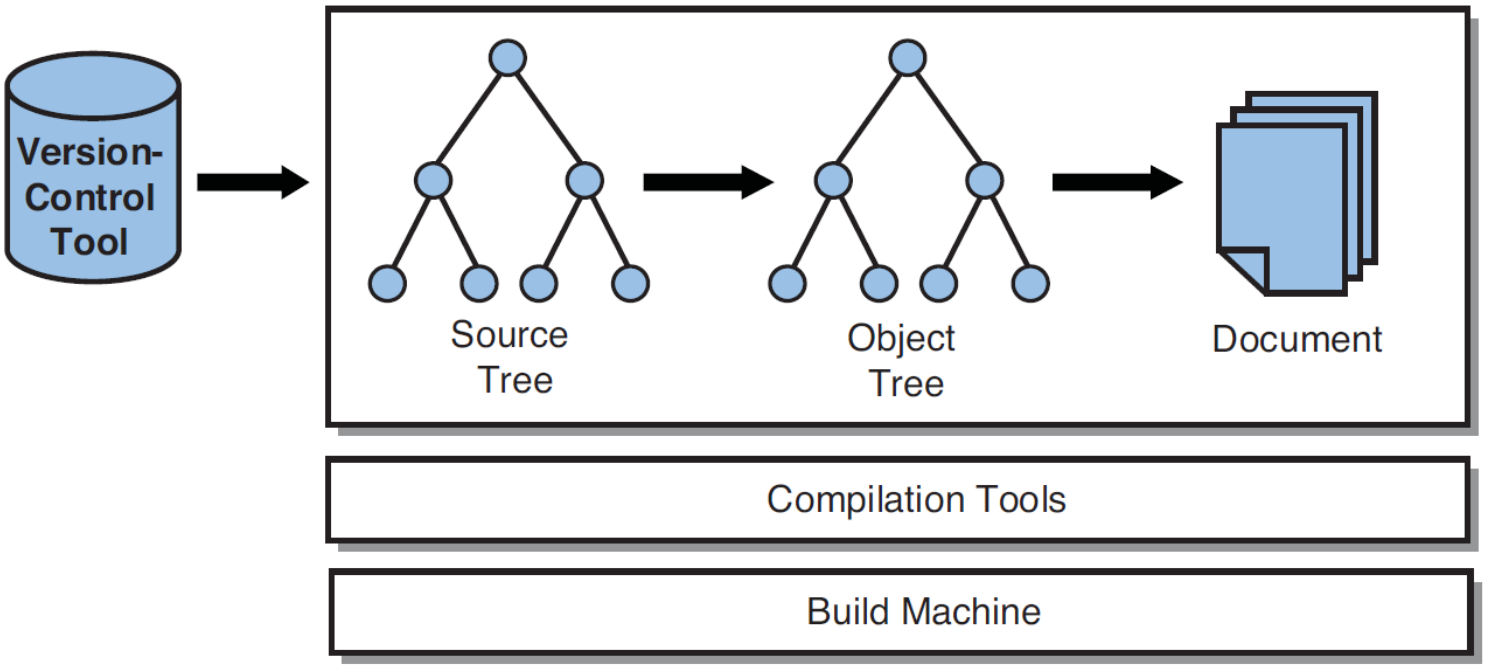
```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      char a[10];
7      a[10] = 0;
8      std::cout << a[10] << std::endl;
9      return 0;
10 }
```

```
$ g++ static_test.cpp -Wall -pedantic
 … No issue reported …
```

```
$ cppcheck  static_test.cpp
Checking static_test.cpp ...
static_test.cpp:7:6: error: Array 'a[10]' accessed at index 10,
which is out of bounds. [arrayIndexOutOfBounds]
    a[10] = 0;
     ^
static_test.cpp:8:16: error: Array 'a[10]' accessed at index 10,
which is out of bounds. [arrayIndexOutOfBounds]
 std::cout << a[10] << std::endl;
                   ^
```

But, won't work with dynamic scenarios, e.g., std::vector!

Documentation Generation



- **Processes source/object tree to generate documentation (e.g. HTML, PDF)**
- **Allows for keeping documentation with source code**
  - **No separate documentation file, easier to maintain**
- **E.g. Doxygen**

Documentation Generation: Doxygen

Live Showcase!

Source code documentation

```
//!  A test class.
/*!
  A more elaborate class description.
*/
 class QTstyle_Test
{
  public:

    //! An enum.
    /*! More detailed enum description. */
    enum TEnum {
                 TVal1, /*!< Enum value TVal1. */
                 TVal2, /*!< Enum value TVal2. */
                 TVal3  /*!< Enum value TVal3. */
               }
         //! Enum pointer.
         /*! Details. */
         *enumPtr,
         //! Enum variable.
         /*! Details. */
         enumVar;

    ....

    //! A normal member taking two arguments and returning an integer value.
    /*!
      \param a an integer argument.
      \param s a constant character pointer.
      \return The test results
      \sa QTstyle_Test(), ~QTstyle_Test(), testMeToo() and publicVar()
    */
    int testMe(int a,const char *s);

....
```

Translation into HTML/LATEX etc.

My Project

Main Page | Classes | Files

QTstyle_Test Class Reference abstract

A test class. More...

`#include <qtstyle_test.hpp>`

Public Types

enum **TEnum** { **TVal1**, **TVal2**, **TVal3** }
An enum. More...

Public Member Functions

QTstyle_Test ()
A constructor. More...

~QTstyle_Test ()
A destructor. More...

int **testMe** (int a, const char *s)
A normal member taking two arguments and returning an integer value. More...

virtual void **testMeToo** (char c1, char c2)=0
A pure virtual member. More...

Public Attributes

enum QTstyle_Test::TEnum * **enumPtr**
Enum pointer. More...

enum QTstyle_Test::TEnum **enumVar**
Enum variable. More...

int **publicVar**
A public variable. More...

int(* **handler**)(int a, int b)
A function variable. More...

Compiler versus Linker

- **Compiler**
  Reads human-written language source files and produces object files that contain a machine code translation of that same program
- **Linker**
  Joins a number of different object files to produce a single executable program.

Compile & Link

```
$ g++ -o hello hello.cpp
$ ./hello
```

Compile

```
$ g++ -c hello.cpp
```

… generates hello.o

Link

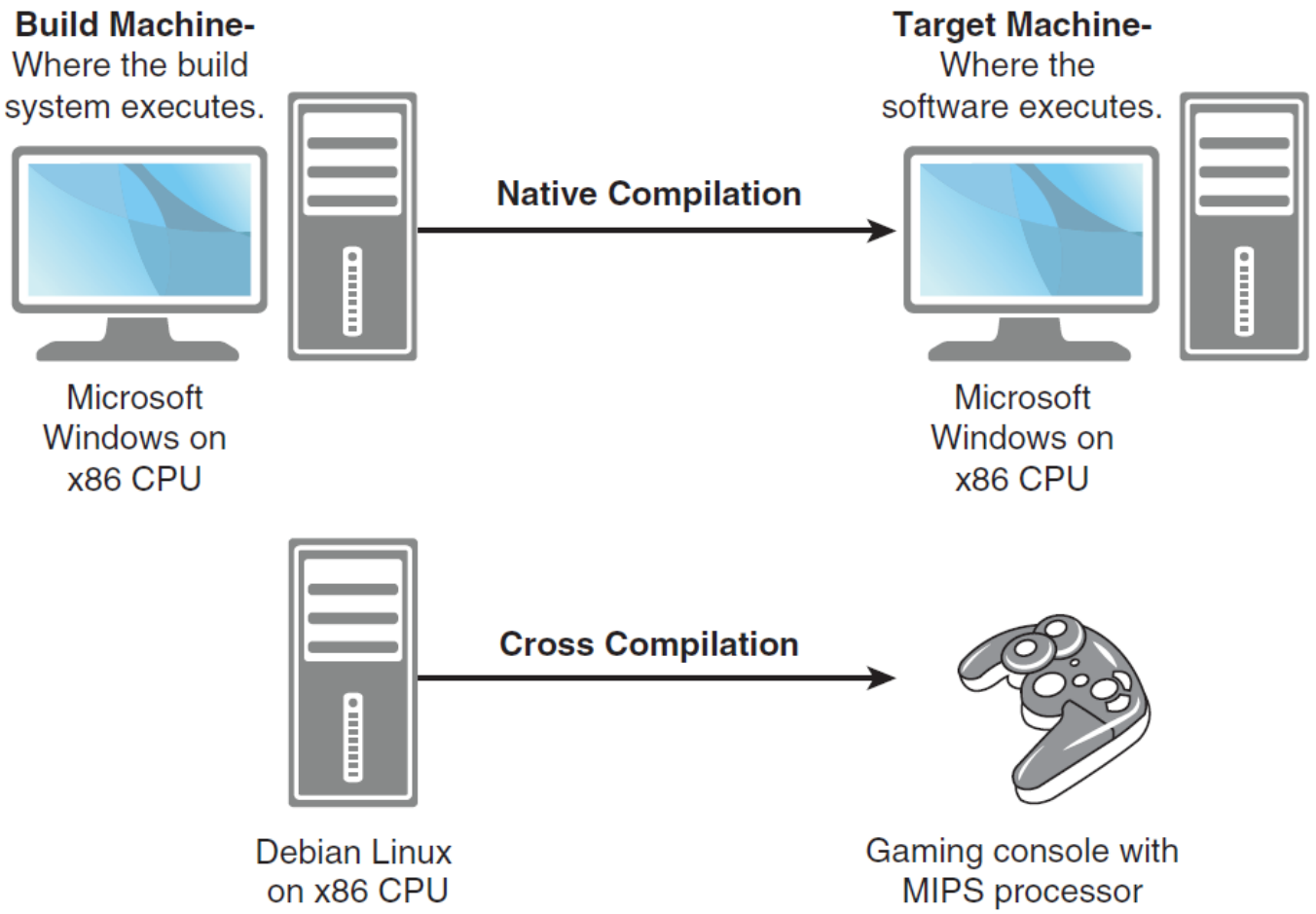
```
$ g++ -g –o hello hello.o
$ ./hello
```

Translator vs Compiler vs Build Tool

- **Translator → Compiler**
  Software that translates text of certain language to another language. If output language is low level language, the translator is called compiler, e.g., g++.

- **Build tool**
  - **Uses compiler and other tools (e.g., documentation and testing tools)**
  - **Orchestrates entire "build" process as efficiently as possible (e.g., parallel compiliation targets)**

Cross-Compilation



e.g. arm-linux-gnu-gcc

Types of Builds

- **Developer (or private) build:**
  checkout source from version control, make changes, compile, make more changes and re-compile.
- **Release build:**
  done by release engineers. Create a complete software package for the software testers to test. When testers are happy the same package goes to customer.
- **Sanity build:**
  similar to release build but does not go to customer. Automated software error checking done several times a day. *aka* “daily build” or “nightly build”

Build System Quality

- Convenience: easy to use
- Correctness: correct compilation/linking of files in specific order
- Performance: must compile as fast as possible (use available parallel resources)
- Scalability: Convenience, correctness, and performance for larger software projects

Runtime View of a Program

- **Executable programs: The sequence of machine-readable instructions that the CPU executes, along with associated data values. This is the fully compiled program that's ready to be loaded into the computer's memory and executed.**

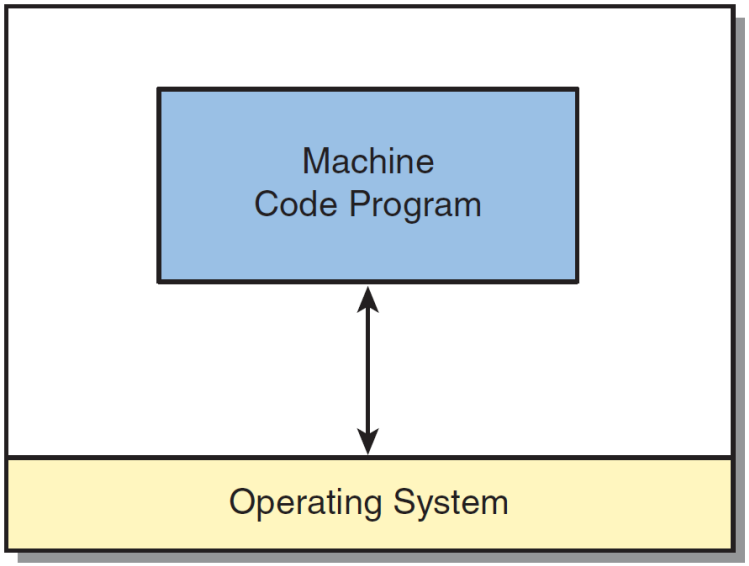
- **Libraries: Collections of commonly used object code that can be reused by different programs. Most operating systems include a standard set of libraries that developers can reuse, instead of requiring each program to provide their own. A library can't be directly loaded and executed on the target machine; it must first be linked with an executable program.**

Runtime View of a Program

- **Configuration and data files: These are not executable files; they provide useful data and configuration information that the program can load from disk.**

- **Distributed programs: This type of software consists of multiple executable programs that communicate with each other across a network or simply as multiple processes running on the same machine. This contrasts with more traditional software that has a single monolithic program image.**

Machine Code Program



The build system fully converted the executable program into the CPU's native machine code. The CPU simply "jumps" to the program's starting location, and all the execution is performed purely using the CPU's hardware. While it's executing, the program optionally makes calls into the operating system to access files and other system resources.

Libraries

- Operating systems ship pre-installed libraries, e.g., file and network I/O, mathematical functions, user interface manipulation etc.
- Software developer can get third-party libraries, e.g., BLAS, Boost
- Or, generate your own:
  - Compile object files
  - Bundle object files into a single library file (static/dynamic)
  - When building an executable, link to the library file to use it's functionality

Static versus Dynamic Libraries

Static Libraries

- Library file consists of collection of object files
- Build-time concept: static library is linked to the executable program during build process
  - Linker determines whether a certain function is required for creating the executable program: extracts appropriate object file from the library and copies it into the executable program.

Dynamic Libraries

- Linker “notes” which libraries are necessary to execute the program
- Upon execution of the program, libraries are loaded into memory as separate entities and then are connected with the main program

Static Library Example in C++

```
// compile source to object files
$ g++ -c sqrt.cpp
$ g++ -c sine.cpp
$ g++ -c cosine.cpp
$ g++ -c tan.cpp

// combine object files into single *.a static library
$ ar -rs mymath.a sqrt.o sine.o cosine.o tan.o

// use static library to build single executable program
$ g++ -c main.cpp
$ g++ -o prog main.o mymath.a
```

Dynamic Library Example in C++

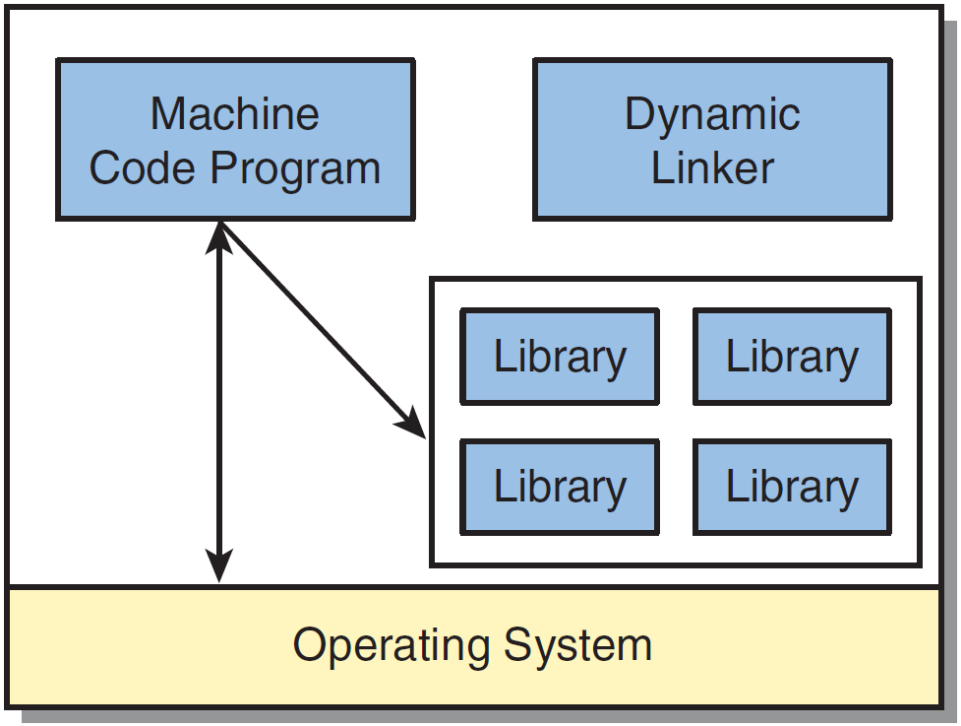
```
// compile source to object files using
// Position Independent Code (PIC)
$ g++ -c -fPIC sqrt.cpp
$ g++ -c -fPIC sine.cpp
$ g++ -c -fPIC cosine.cpp
$ g++ -c -fPIC tan.cpp
```

@PIC: required so that the object files can be loaded at any memory location the program requires

```
// combine object files into single *.so dynamic library
$ g++ -shared -o libmymath.so sqrt.o sine.o cosine.o tan.o

// use static library to build single executable program
$ g++ -c main.cpp
$ g++ -o prog main.o -L. -lmymath

$ export LD_LIBRARY_PATH=.
$ ldd prog
linux-gate.so.1 => (0xffffe000)
libmymath.so => ./libmymath.so (0xb80a7000)
...
```

Static versus Dynamic Libraries

- **Dynamic linking is more complex than static, but:**
  - **Possible to upgrade to a newer version of a library without re-building the program.**
  - **Many operating systems can optimize their memory usage by loading only a single copy of the library into memory, yet sharing it with other programs that require that same library.**



C++ Compilers and File Types

- **Many C++ compilers exist, e.g., g++, Clang, Microsoft/Intel C++ compiler, etc.**
- **Focus on g++ and Linux**
- **g++ uses a *toolchain* approach for compiling**
  - **Preprocessor, expands macro definitions**
  - **Compiler, translates source code into assembly language**
  - **Assembler, translates assembly language into object files**
  - **Linker, joins object files into single executable program**

Build Tools – GNU Make

- Make has been around since 1977
- Considered the first build tool
- Until today an easy and important first step into build tools
- Central aspect is the “rule”

```
myprog: prog.cpp lib.cpp
        g++ -o myprog prog.cpp lib.cpp
```

- If time stamp of `prog.cpp` or `lib.cpp` is more recent than `myprog`, `myprog` is rebuilt.
- Build description is stored in `Makefiles`: holds all rules and dependencies

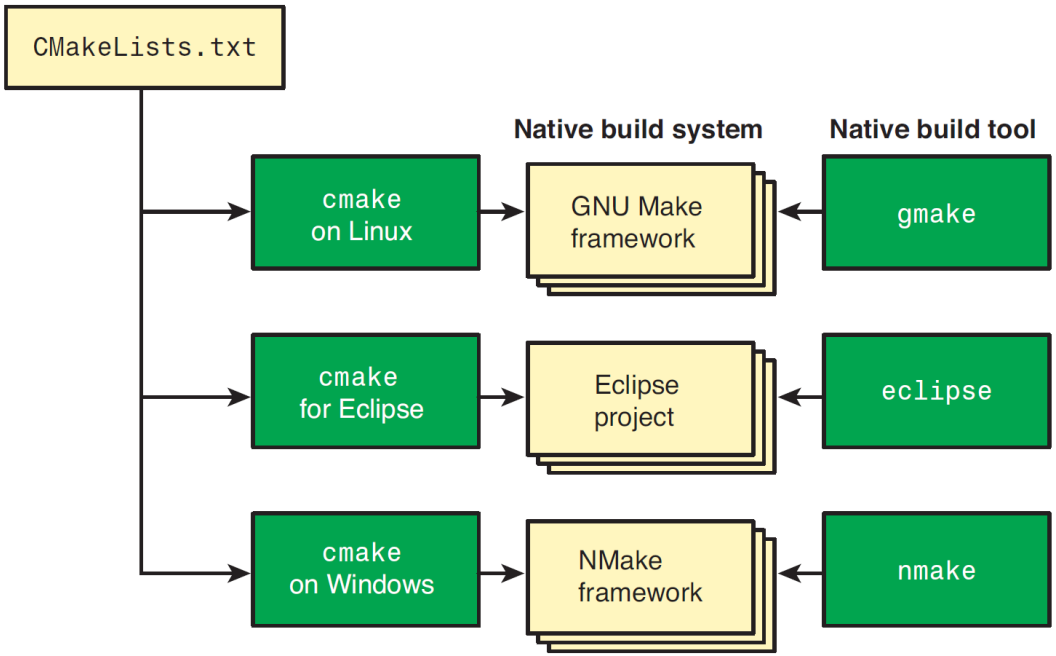
Example

```
SRCS = calc.cpp add.cpp sub.cpp mult.cpp
PROG = calculator
CC = g++
CFLAGS = -g
OBJS = $(SRCS:.cpp=.o) # same filenames as source but with *.o
$(PROG): $(OBJS)
        $(CC) $(CFLAGS) -o $@ $^
$(OBJS): numbers.h
```

- `$@` … contains the filename of the current rule's target.
- `$^` … lists all source files

Build Tools – CMake

- CMake doesn't actually execute the build
- Translates higher-level build description (`CMakeLists.txt`) into a lower-level format accepted by other tools, e.g., Make
- Simplifies construction of large build systems
- Support cross-platform builds



Q4: Do you know a platform independent build system?

Build Tools – Cmake: Simple Example

CMakeLists.txt **Live Showcase!**

```
project (hello)
cmake_minimum_required(VERSION 3.16.3)
add_executable(hello helloworld.cpp)
```

```
$ mkdir build && cd build
$ cmake ..
-- The C compiler identification is GNU 9.3.0
 -- The CXX compiler identification is GNU 9.3.0
 -- Check for working C compiler: /usr/bin/gcc
 -- Check for working C compiler: /usr/bin/gcc -- works
 -- Detecting C compiler ABI info
 -- Detecting C compiler ABI info - done
 -- Check for working CXX compiler: /usr/bin/c++
 -- Check for working CXX compiler: /usr/bin/c++ -- works
 -- Detecting CXX compiler ABI info
 -- Detecting CXX compiler ABI info - done
 -- Configuring done
 -- Generating done
 -- Build files have been written to:
    /home/folder/cmake/helloworld

$ make
[ 50%] Building CXX object CMakeFiles/hello.dir/helloworld.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
$./hello
```

Build Tools – CMake

```
project (my_project_name)
cmake_minimum_required (VERSION 2.6)
set (wife Grace)
set (dog Stan)
message ("${wife}, please take ${dog} for a walk")
set_property (SOURCE add.cpp PROPERTY Author Peter)
set_property (SOURCE mult.cpp PROPERTY Author John)
get_property (author_name SOURCE add.cpp PROPERTY Author)
message("The author of add.cpp is ${author_name}")
```

- `command ( arg1 arg2 ... )`

Build Tools – CMake

Set compiliation flags

```
set (CMAKE_BUILD_TYPE Debug)  # on Unix: -g
```

Create an executable

```
add_executable (calculator calc add sub mult)
```

Create a library

```
add_library (math STATIC add sub mult)
add_executable (calculator calc)
target_link_libraries (calculator math)
```

Cross-platform library finding

```
find_library (LIB_MATH_PATH m /usr/local/lib /usr/lib64)
message ("The path to the math library is ${LIB_MATH_PATH}")
```

CMake Tutorials

https://cmake.org/cmake/help/latest/guide/tutorial/index.html

Outline

- Quiz Wrapup
- Build Systems
- **Debugging**
- Next Quiz

Motivation

- Debugging essential for finding and eliminating bugs
- Developer must have an option to monitor progress of the code as it executes
- For efficient debugging, compiler needs to generate debugging information when generating the executable program
- Compile your code with the `-g` flag
- Then use a debugging software of your choice
  - gdb – most basic but works in terminal and is widely available
  - Other GUI tools are available: Code::Blocks, Codelite, DDD, Eclipse, KDevelop, Nemiver, Visual Studio, Qt Creator, NetBeans,

Q5: What is gdb?

gdb Example

```
$ g++ -g -o prog prog.cpp
$ gdb prog
...
(gdb) list
1 #include <iostream>
2
3 int main()
4 {
5   int i;
6
7   for (i = 0; i != 100; i++) {
8     std::cout << "The next number is " << i << std::endl;
9   }
10 return 0; }
```

gdb Example

Live Showcase!

```
(gdb) break 7
Breakpoint 1 at 0x8048435: file prog.cpp, line 7.

(gdb) run
Starting program: /home/psmith/Book/examples/debugging-
session/prog
Breakpoint 1, main (argc=<value optimized out>,
argv=<value optimized out>) at prog.cpp:7
7 for (i = 0; i != 100; i++) {

(gdb) next
8 std::cout << "The next number is " << i << std::endl;

(gdb) next
The next number is 0
7 for (i = 0; i != 100; i++) {

(gdb) next
8 std::cout << "The next number is " << i << std::endl;

(gdb) print i
$1 = 1
```

Memory Checking

- gdb, as a debugger, will help you find many bugs
- A more deeper analysis provides a memory checker, e.g., Valgrind
- Valgrind won't let you step through a program, but:
  - Checks for uninitialized values
  - Over/underflowing dynamic memory
  - Determines the cause of segfaults
  - Many other features, e.g., memory consumption throughout execution
  - However, program execution 10-30 times slower than normal

Valgrind Example

```
void f(void)
{
    int* x = malloc(10*sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
    return 0;
}
```

```
valgrind-tut $ gcc -O0 -g -Wall -o a.x a.c
valgrind-tut $ valgrind ./a.x
==13754== Memcheck, a memory error detector
==13754== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==13754== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==13754== Command: ./a.x
==13754==
==13754== Invalid write of size 4
==13754==    at 0x80483FF: f (a.c:6)
==13754==    by 0x8048411: main (a.c:11)
==13754==  Address 0x41f1050 is 0 bytes after a block of size 40 alloc'd
==13754==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==13754==    by 0x80483F5: f (a.c:5)
==13754==    by 0x8048411: main (a.c:11)
==13754==
==13754==
==13754== HEAP SUMMARY:
==13754==     in use at exit: 40 bytes in 1 blocks
==13754==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==13754==
==13754== LEAK SUMMARY:
==13754==    definitely lost: 40 bytes in 1 blocks
==13754==    indirectly lost: 0 bytes in 0 blocks
==13754==      possibly lost: 0 bytes in 0 blocks
==13754==    still reachable: 0 bytes in 0 blocks
==13754==         suppressed: 0 bytes in 0 blocks
==13754== Rerun with --leak-check=full to see details of leaked memory
==13754==
==13754== For counts of detected and suppressed errors, rerun with: -v
==13754== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
valgrind-tut $
```

Outline

- Quiz Wrapup
- Build Systems
- Debugging
- **Next Quiz**

New Quiz

1. What is the `Makefile` variable which lists all source files?
2. Create a simple `Makefile` for the following scenario:
   - 5 source files named `1-5.cpp`
   - Must be linked to the dynamic `blas` library
   - Must use an optimized compilation level of 2
   - Compiler flags must be set via a single parameter
3. What is the difference between Make and CMake?
4. Do you know a typical version control software for Unix development?
5. Do you know an open source license and its key properties?