

# CS1050

Makefiles and Automated Build Systems

# Automated Build Systems

- It is tedious to manually compile and link programs, especially if you have multiple files needed to “build” your program.
- Even compiling a single file program as we usually do, often requires special compiler switches.
- CS 1050 gets around this via a script file called compile:  

```
gcc "$@" -g -Wall -Werror -lm -std=c11
```
- To make it easier to compile and link programs, many people use an automated build system. Perhaps the most popular automated build system of all time is “make”.

# Example complex program

- JimR.h:

```
void JimRCoolFunction();
```

```
void JimRCoolerFunction();
```

# Example complex program

- JimRCoolFunction.c:

```
#include <stdio.h>
void JimRCoolFunction()
{
    printf("This is a very cool function.\n");
}
```

# Example complex program

- JimRCoolerFunction.c:

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

```
void JimRCoolerFunction()
```

```
{
```

```
    printf("This function is much cooler than the  
other one.\n");
```

```
}
```

# Example complex program

- main.c:

```
#include <stdio.h>
#include "JimR.h"
int main(void)
{
    JimRCoolFunction();
    JimRCoolerFunction();
}
```

# Building the example complex program

```
cc -c -g -Wall -Werror -lm -std=c11 JimRCoolerFunction.c
```

```
cc -c -g -Wall -Werror -lm -std=c11 JimRCoolFunction.c
```

```
cc -c -g -Wall -Werror -lm -std=c11 main.c
```

```
cc main.o JimRCoolFunction.o JimRCoolerFunction.o
```

- We could put those commands into a “shell script”, but . . .(see next slide)

# But, what if I change something?

- Just rebuild the whole thing?
- Just compile the pieces I need?
- What if the program is even more complex, with hundreds of source files needed?



# man make

## NAME

make - GNU make utility to maintain groups of programs

## SYNOPSIS

make [ -f makefile ] [ options ] ... [ targets ] ...

## DESCRIPTION

The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. The manual describes the GNU implementation of make, which was written by Richard Stallman and Roland McGrath, and is currently maintained by Paul Smith. Our examples show C programs, since they are most common, but you can use make with any programming language whose compiler can be run with a shell command. In fact, make is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

To prepare to use make, you must write a file called the makefile that describes the relationships among files in your program, and states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.

# man make (cont.)

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The make program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

make executes commands in the makefile to update one or more target names, where name is typically a program. If no -f option is present, make will look for the makefiles GNUmakefile, makefile, and Makefile, in that order.

Normally you should call your makefile either makefile or Makefile. (We recommend Makefile because it appears prominently near the beginning of a directory listing, right near other important files such as README.) The first name checked, GNU-makefile, is not recommended for most makefiles. You should use this name if you have a makefile that is specific to GNU make, and will not be understood by other versions of make. If makefile is '-', the standard input is read.

make updates a target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exist.

# Basic recipe for a simple makefile

Target: dependencies

[tab]system command

[tab]system command

....

[blank line]

# Back to our example: Makefile

a.out : main.o JimRCoolFunction.o JimRCoolerFunction.o

cc main.o JimRCoolFunction.o JimRCoolerFunction.o -o a.out

main.o : main.c

cc -c -g -Wall -Werror -lm -std=c11 main.c

JimRCoolFunction.o : JimRCoolFunction.c

cc -c -g -Wall -Werror -lm -std=c11 JimRCoolFunction.c

JimRCoolerFunction.o : JimRCoolerFunction.c

cc -c -g -Wall -Werror -lm -std=c11 JimRCoolerFunction.c

# Macros

- Sure is a lot of duplication. We have to use `-Wall`, `-Werror`, etc. for every file we compile.
- Answer: Macros!

MACRO = whatever you want it to equal

- To reference it:  
\$(MACRO)

# Example

OBJS = main.o JimRCoolFunction.o JimRCoolerFunction.o

CC = gcc

CFLAGS = -Wall -Werror -c -g -std=c11

LDFLAGS = -lm

a.out : \$(OBJS)

cc \$(OBJS) \$(LDFLAGS) -o a.out

main.o : main.c

cc \$(CFLAGS) main.c

JimRCoolFunction.o : JimRCoolFunction.c

cc \$(CFLAGS) JimRCoolFunction.c

JimRCoolerFunction.o : JimRCoolerFunction.c

cc \$(CFLAGS) JimRCoolerFunction.c

# Cleaning up

- Sometimes you want to rebuild everything
- Convention is to have a special target called “clean” that depends on nothing.

clean :

rm -f a.out

rm -f \$(OBJS)

# Automatic Variables

- Automatic variables are used in the action steps for convenience
- \$@ - target name
- \$< - first dependency
- \$^ - all dependencies



# Example

a.out : \$(OBJS)

cc \$(OBJS) \$(LDFLAGS) -o \$@

main.o : main.c

cc \$(CFLAGS) \$^

# Pattern Rules

- Well, things are better, but we still have a lot of duplication.
- Every time we want to compile a file we have to put in the object file and the source file (which differ only by .c versus .o extension), and then the compilation action.
- Answer: Pattern Rules!

`%.o : %.c`

`cc $(CFLAGS) $^`

# Example

OBJS = main.o JimRCoolFunction.o JimRCoolerFunction.o

CC = gcc

CFLAGS = -Wall -Werror -c -g -std=c11

LDFLAGS = -lm

%.o : %.c

cc \$(CFLAGS) \$^

a.out : \$(OBJS)

cc \$(OBJS) \$(LDFLAGS) -o \$@

# Make references

- <https://www.gnu.org/software/make/manual/make.html>
- <http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/>
- <http://mrbook.org/blog/tutorials/make/>

# Automated Build Systems in Practice

- Autotools
- CMake
- IDE-driven dependency systems like Visual Studio “solution files”

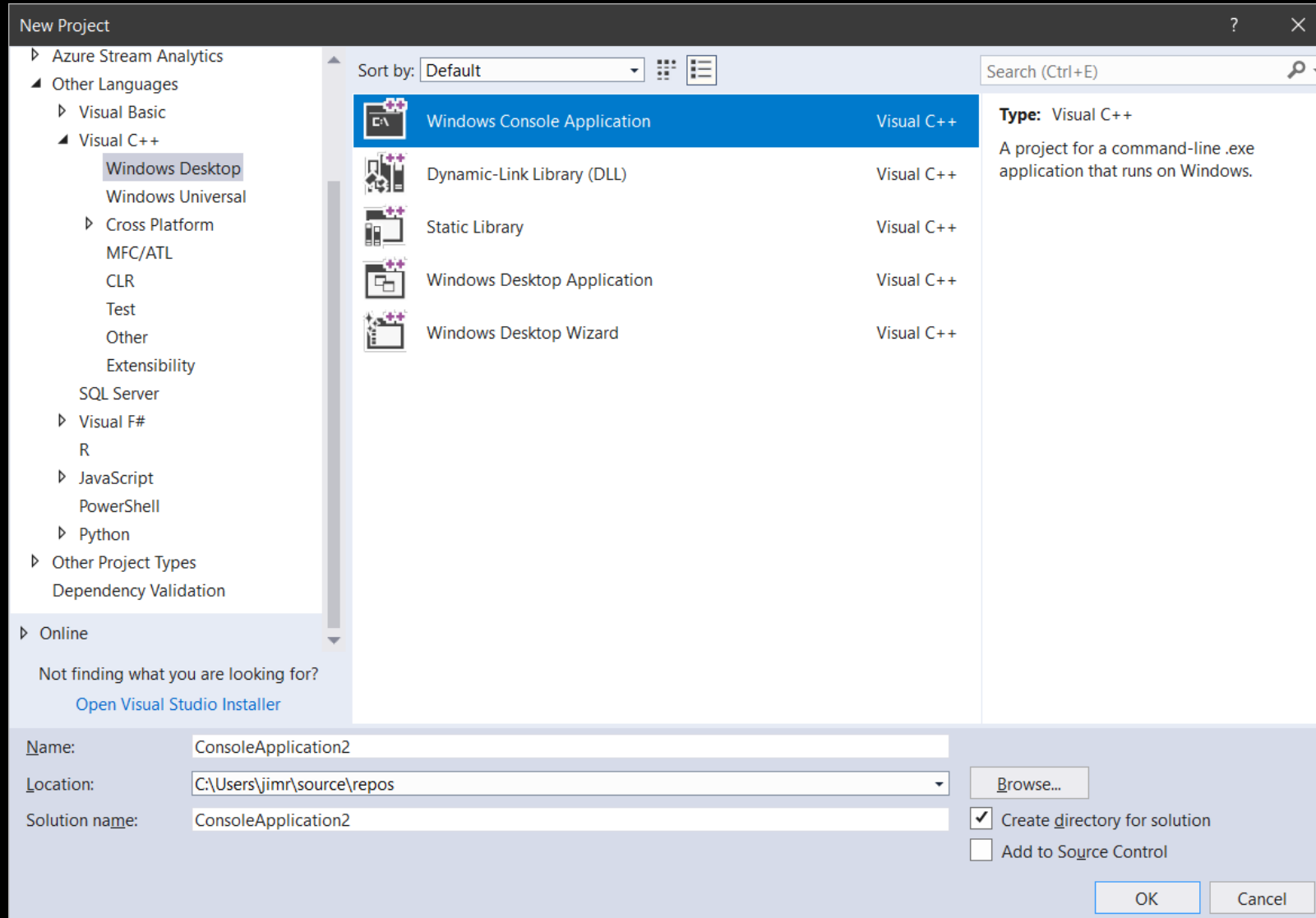
# Autotools

- [https://www.gnu.org/software/automake/manual/html\\_node/Autotools-Introduction.html](https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html)

# CMake

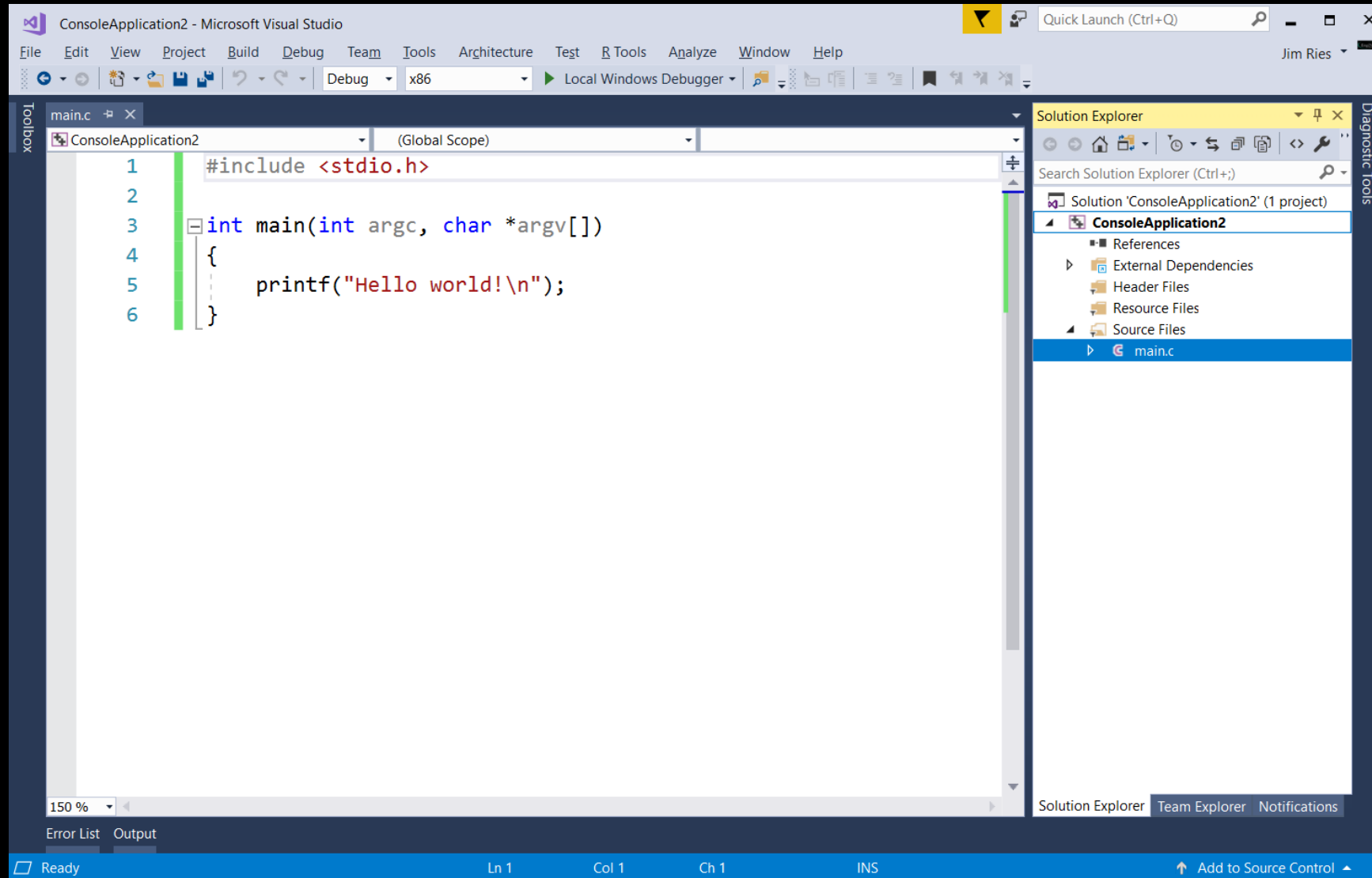
- <https://cmake.org/cmake/help/v3.12/>

# Visual Studio

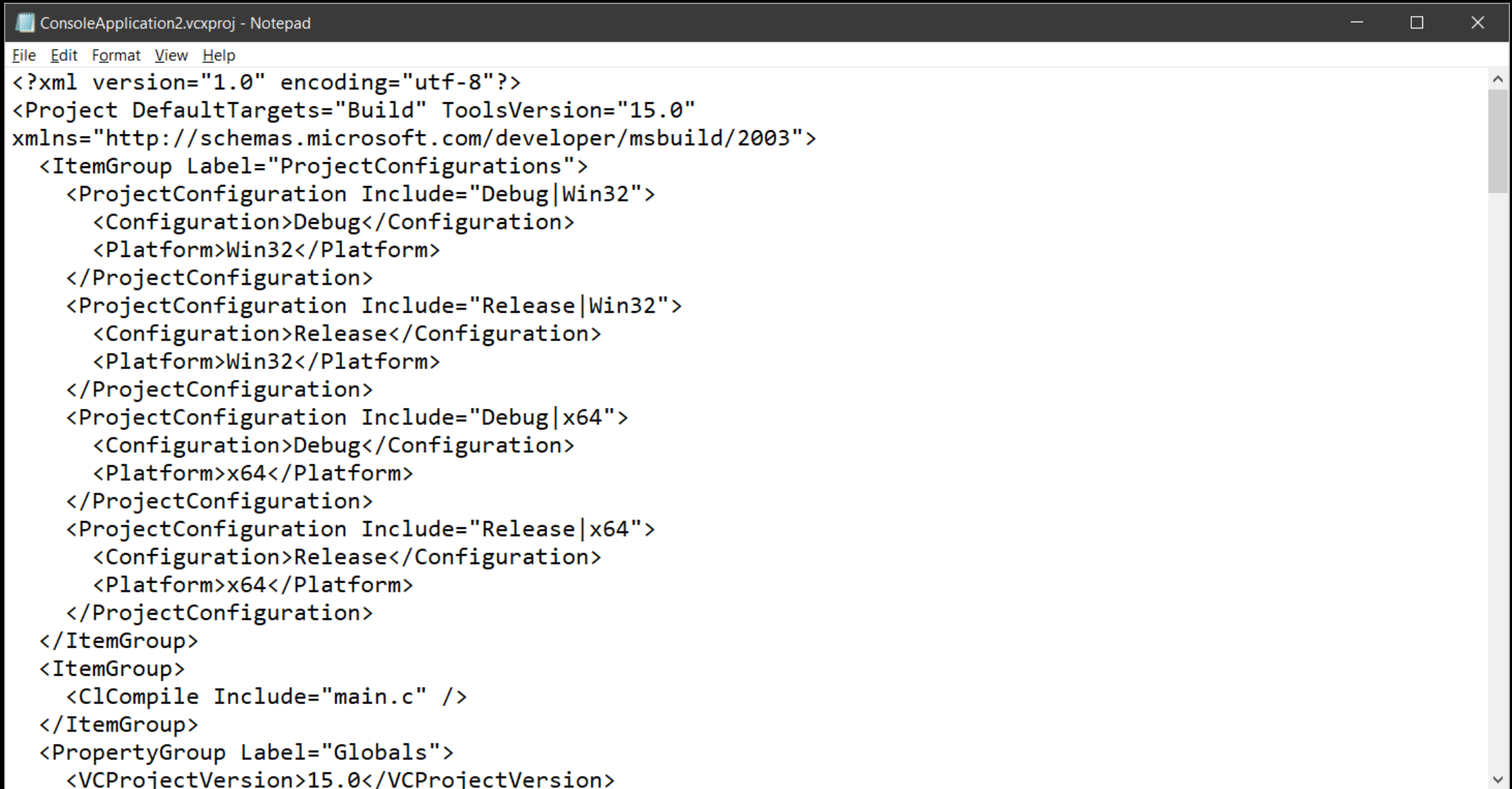




# Visual Studio



# Visual Studio



The image shows a Notepad window titled "ConsoleApplication2.vcxproj - Notepad". The window contains an XML file with the following content:

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="15.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup Label="ProjectConfigurations">
    <ProjectConfiguration Include="Debug|Win32">
      <Configuration>Debug</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Release|Win32">
      <Configuration>Release</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Debug|x64">
      <Configuration>Debug</Configuration>
      <Platform>x64</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Release|x64">
      <Configuration>Release</Configuration>
      <Platform>x64</Platform>
    </ProjectConfiguration>
  </ItemGroup>
  <ItemGroup>
    <ClCompile Include="main.c" />
  </ItemGroup>
  <PropertyGroup Label="Globals">
    <VCProjectVersion>15.0</VCProjectVersion>
```