NOTE:

The following text was taken from

https://solarianprogrammer.com/2017/05/21/compiling-gcc-macos/

I reproduce it here, so that I won't loose it.

I used it to build gcc 9.2 and all went fine.

Compiling GCC 9 on macOS Mojave

Posted on May 21, 2017 by Paul

Updated 4 May 2019

In this tutorial, I will show you how to compile from source and install the current stable version of GCC with Graphite loop optimizations on your macOS computer. The instructions from this tutorial were tested with Xcode 10 and Mojave (macOS 10.14).

Clang, the default compiler for macOS, supports only C, C++, Objective-C and Objective-C++. If you are interested in a modern Fortran compiler, e.g. you will need *gfortran* that comes with GCC. Another reason to have the latest stable version of GCC on your macOS is that it provides you with an alternative C and C++ compiler. Testing your code with two different compilers is always a good idea.

Building *GCC 9* from sources could take some time, in my case it took about two hours on a MacBook Air with a 16GB of RAM. If you want to avoid the wait time or if you have any problem building from source, you can download my binary version.

In order to compile GCC from sources you will need a working C++ compiler. In the remaining of this article I will assume that you have installed the Command Line Tools for Xcode. At the time of this writing Apple's Command Line Tools maps the *gcc* and *g++* to *clang* and *clang++*. If you don't have the Command Line Tools installed, open a Terminal and write:

```
1 xcode-select --install
```

which will guide through the installation process.

macOS Mojave changed the location of the system headers, this broke the *GCC 9* build process. In order to build *GCC* install the required header files in the old location:

```
1 cd /Library/Developer/CommandLineTools/Packages/
2 open .
```

A window should open, double click the existing *pkg* file, and accept the defaults. This should solve the problem of the missing header files.

Let's start by creating a working folder:

```
1 cd ~
2 mkdir gcc_all && cd gcc_all
```

Next, we can download and extract the latest stable version of *GCC*:

```
1 curl -L https://ftpmirror.gnu.org/gcc/gcc-9.1.0/gcc-9.1.0.tar.xz | tar xf
-
```

GCC 9 depends on a couple of other libraries that can be downloaded with:

```
1 curl -L ftp://gcc.gnu.org/pub/gcc/infrastructure/gmp-6.1.0.tar.bz2 | tar
xf -
```

```
2 curl -L ftp://gcc.gnu.org/pub/gcc/infrastructure/mpfr-3.1.4.tar.bz2 | tar
xf -
3 curl -L ftp://gcc.gnu.org/pub/gcc/infrastructure/mpc-1.0.3.tar.gz | tar
xf -
4 curl -L ftp://gcc.gnu.org/pub/gcc/infrastructure/isl-0.18.tar.bz2 | tar
xf -
```

We will start by compiling and installing the *gmp* library:

```
1 cd gmp*
2 mkdir build && cd build
3 ../configure --prefix=/usr/local/gcc-9.1 --enable-cxx
4 make -j 8
5 sudo make install-strip
```

We will do the same steps for *mpfr* now:

```
1 cd ../..
2 cd mpfr*
3 mkdir build && cd build
4 ../configure --prefix=/usr/local/gcc-9.1 --with-gmp=/usr/local/gcc-9.1
5 make -j 8
6 sudo make install-strip
```

Now, we are going to build *mpc*:

```
1 cd ../..
2 cd mpc*
3 mkdir build && cd build
4 ../configure --prefix=/usr/local/gcc-9.1 \
5              --with-gmp=/usr/local/gcc-9.1 \
6              --with-mpfr=/usr/local/gcc-9.1
7 make -j 8
8 sudo make install-strip
```

Next step is to build the library for the Graphite loop optimizations:

```
1 cd ../..
2 cd isl*
3 mkdir build && cd build
4 ../configure --prefix=/usr/local/gcc-9.1 --with-gmp-
prefix=/usr/local/gcc-9.1
5 make -j 8
6 sudo make install-strip
```

We are ready to compile *GCC* now. Be prepared that this could take more than one hour on some machines … Since I'm interested only in the C, C++ and Fortran compilers, this is the configure command I've used on my machine:

```
cd ../..
cd gcc*
mkdir build && cd build
../configure --prefix=/usr/local/gcc-9.1 \
             --enable-checking=release \
             --with-gmp=/usr/local/gcc-9.1 \
             --with-mpfr=/usr/local/gcc-9.1 \
             --with-mpc=/usr/local/gcc-9.1 \
             --enable-languages=c,c++,fortran \
             --with-isl=/usr/local/gcc-9.1 \
             --program-suffix=-9.1
```

The above command instructs the configure app where we have installed gmp, mpfr, mpc and isl; also it tells to add a prefix to all the resulting executable programs, so for example if you will invoke GCC 9.1.0 you will write *gcc-9.1*, the *gcc* command will invoke Apple's version of *clang*.

If you are interested in building more compilers available in the GCC collection modify the *–enable-languages* configure option.

And now, the final touches:

```
make -j 8
```

Grab a coffee, maybe a book, and wait … this should take approximately, depending on your computer configuration, an hour … or more … and about 5.24GB of your disk space for the build folder.

Install the compiled gcc in */usr/local/gcc-9.1*:

```
sudo make install-strip
```

Now, you can keep the new compiler completely isolated from your Apple's gcc compiler and, when you need to use it, just modify your path by writing in Terminal:

```
export PATH=/usr/local/gcc-9.1/bin:$PATH
```

If you want to avoid writing the above command each time you open a Terminal, save the above command in the file *.bash_profile* from your Home folder, e.g:

```
echo 'export PATH=/usr/local/gcc-9.1/bin:$PATH' >> ~/.bash_profile
source ~/.bash_profile
```

You should be able to invoke any of the newly compiled compilers C, C++, Fortran …, invoking g++ is as simple as writing in your Terminal:

```
g++-9.1 test.cpp -o test
```

Remember to erase the working folder from your *HOME* if you want to recover some space:

```
cd ~
rm -rf gcc_all
```

Next, I'll show you how to check if the compiler was properly installed by compiling and running a few examples. GCC 9 uses by default the C++14 standard and C11 for the C coders, you should be able to compile any valid C++14 code directly. In your favorite text editor, copy and save this test program (I'll assume you will save the file in your Home directory):

```
//Program to test the C++ lambda syntax and initializer lists
#include <iostream>
```

```
 3 #include <vector>
 4
 5 int main()
 6 {
 7   // Test lambda
 8   std::cout << [](int m, int n) { return m + n;} (2,4) << '\n';
 9
10   // Test initializer lists and range based for loop
11   std::vector<int> V{1,2,3};
12
13   std::cout << "V =\n";
14   for(auto e : V) {
15     std::cout << e << '\n';
16   }
17
18   return 0;
19 }
```

Compiling and running the above lambda example:

```
1 g++-9.1 tst_lambda.cpp -o tst_lambda
2 ./tst_lambda
3 6
4 V =
5 1
6 2
7 3
```

We could also compile a C++ code that uses threads:

```
 1 //Create a C++ thread from the main program
 2
 3 #include <iostream>
 4 #include <thread>
 5
 6 //This function will be called from a thread
 7 void call_from_thread() {
 8     std::cout << "Hello, World!\n";
 9 }
10
11 int main() {
```

```
12      //Launch a thread
13      std::thread t1(call_from_thread);
14
15      //Join the thread with the main thread
16      t1.join();
17
18      return 0;
19  }
```

Next, we present a simple C++ code that uses regular expressions to check if the input read from stdin is a floating point number:

```
 1  //Uses a regex to check if the input is a floating point number
 2
 3  #include <iostream>
 4  #include <regex>
 5  #include <string>
 6
 7  int main()
 8  {
 9    std::string input;
10    std::regex rr("((\\+|-
)?[[:digit:]]+)(\\.(([[:digit:]]+)?))?((e|E)((\\+|-)?)[[:digit:]]+)?");
11    //As long as the input is correct ask for another number
12    while(true)
13    {
14      std::cout << "Give me a real number!\n";
15      std::cin >> input;
16
17      if(!std::cin) break;
18
19      //Exit when the user inputs q
20      if(input == "q") {
21          break;
22      }
23
24      if(regex_match(input,rr)) {
25        std::cout << "float\n";
26      } else {
27        std::cout<<"Invalid input\n";
```

```
28      }
29    }
30 }
```

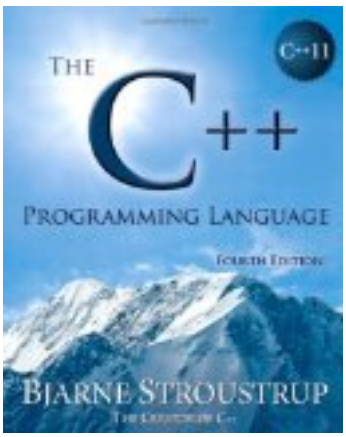
If you are a Fortran programmer, you can use some of the Fortran 2008 features like *doconcurrent* with gfortran-9.1:

```
 1 integer,parameter::mm=100000
 2 real::a(mm), b(mm)
 3 real::fact=0.5
 4
 5 ! initialize the arrays
 6 ! ...
 7
 8 do concurrent (i = 1 : mm)
 9       a(i) = a(i) + b(i)
10 enddo
11
12 end
```

The above code can be compiled with (assuming you've named it tst_concurrent_do.f90):

```
1 gfortran-9.1 tst_concurrent_do.f90 -o tst_concurrent_do
2 ./tst_concurrent_do
```

If you are interested in learning more about the new C++11/C++14 syntax I would recommend reading The C++ Programming Language by Bjarne Stroustrup.



or, Professional C++ by M. Gregoire, N. A. Solter, S. J. Kleper 2nd edition:

If you need to brush your Fortran knowledge a good book is Modern Fortran Explained by M. Metcalf, J. Reid and M. Cohen: