Introduction to Compiled Languages

Applied Math 205 Group Activity

09-Sep-2021

Presented by:

Michael S. Emanuel*, Chris Rycroft

Outline

1. Introduction and Motivation
2. Hello World (With Style)
3. Data Structures (RPG Characters)
4. Functions (Square Root)
5. Pointers and Arrays (Matrix Multiplication)
6. Classes (Timer)
7. Build Automation (GNU Make)

Introduction and Motivation

Introductions: Who am I?

- Grew up in New Rochelle, NY
  - Oldest of 5 children
- Married, father of three
  - Son 7, daughters 4 & 1
- Harvard class of 1999 (mathematics)
- Worked for 17 years in finance
- G2 in the Applied Math department
- Class distance runner in younger days
  - 90th in 2011 Boston Marathon 2:29:52

Emanuel Family Vacation; July 2021, Brewster MA
L-R: Renee, Christie, Victor, Michael, Ruth.

Who are You?

- Everyone in the room please take a turn
- Tell us your name and academic program
- Please share at least three things about you outside of school

Audience Poll

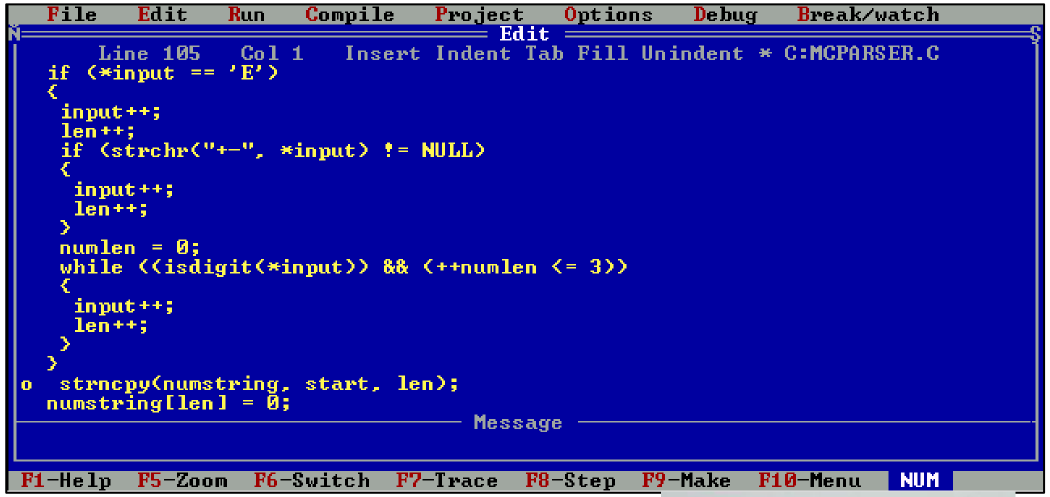
- Please raise your hand if …
- You are familiar with programming in Python or similar language
- You have written at least a “hello world” program in C/C++
- You are familiar with version control / git
- You are familiar with make and build automation tools
- You have written a “serious” C/C++ program before

Why Learn C / C++?

- Isn’t Python the hot language these days?
- Isn’t C / C++ notoriously hard to learn and painful to debug?
- Many people would say “yes” to both! But…
- C++ is **FAST**. In many applications, that is crucial.
- C++ is **highly expressive** and lets you get **close to the hardware**
- This is a course on **scientific computing,** where **C++ is popular**
- C and C++ have stood the **test of time**

Personal Reminiscence

- I first learned to program a computer circa 1991
- I used an early version of the Borland Turbo C++ compiler for MS-DOS
- I wrote programs to do text conversions used in typesetting for my dad’s publishing business
- I later programmed Tetris for fun
- The point of this anecdote is that C has been useful for a long time!



Top: Screenshot of Borland Turbo C 2.01 (DOS)
Similar to the IDE I learned to program C on.
Bottom: Floppy disks and manual for Borland C++ in this era.

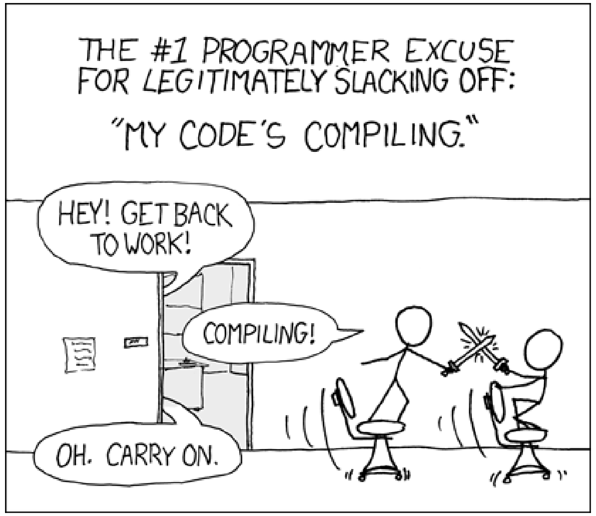
Compiled vs. Interpreted Languages

- What’s the difference between compiled and interpreted languages?
- Compiled Language:
  - You write a program in source code, e.g. C++
  - You run a special program called a **compiler**, e.g. gcc or g++
  - This creates an executable program – machine language that can run on your hardware and operating system
- Interpreted Language:
  - You write a program or just one line of source code, e.g. Python
  - You run a special program called an interpreter, e.g. python
  - This interpreter program runs each line of code, one at a time
  - The interpreter itself is an executable program that was compiled…
  - For example, does anyone know what language the python interpreter is written in?

Comparison of Compiled / Interpreted

Compiled Languages

- Faster at runtime
- Often slower to write program
- Closer to hardware
- Separate build phase

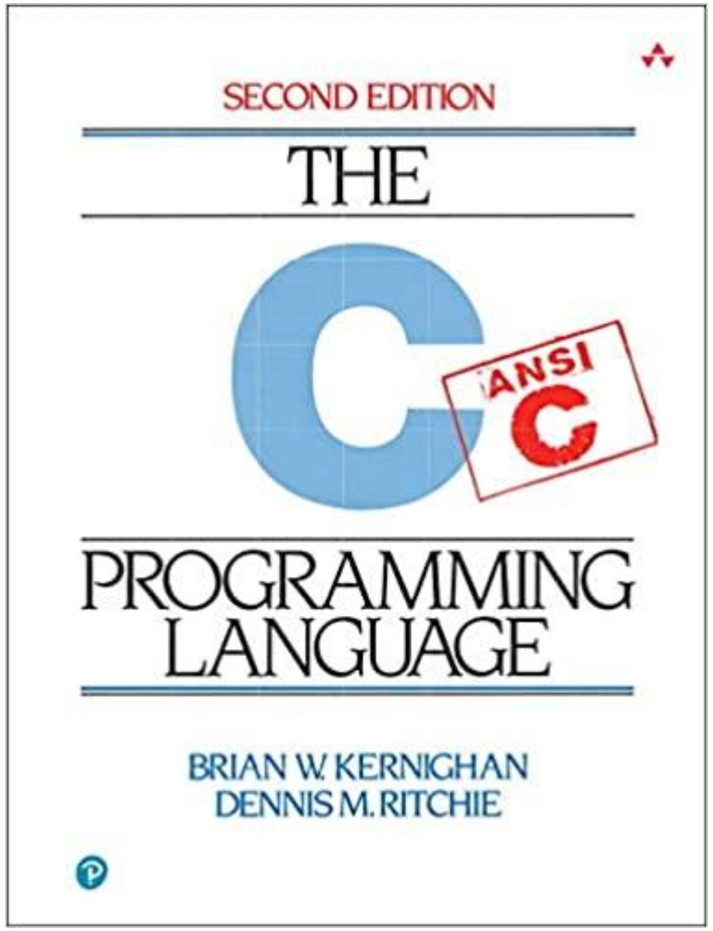


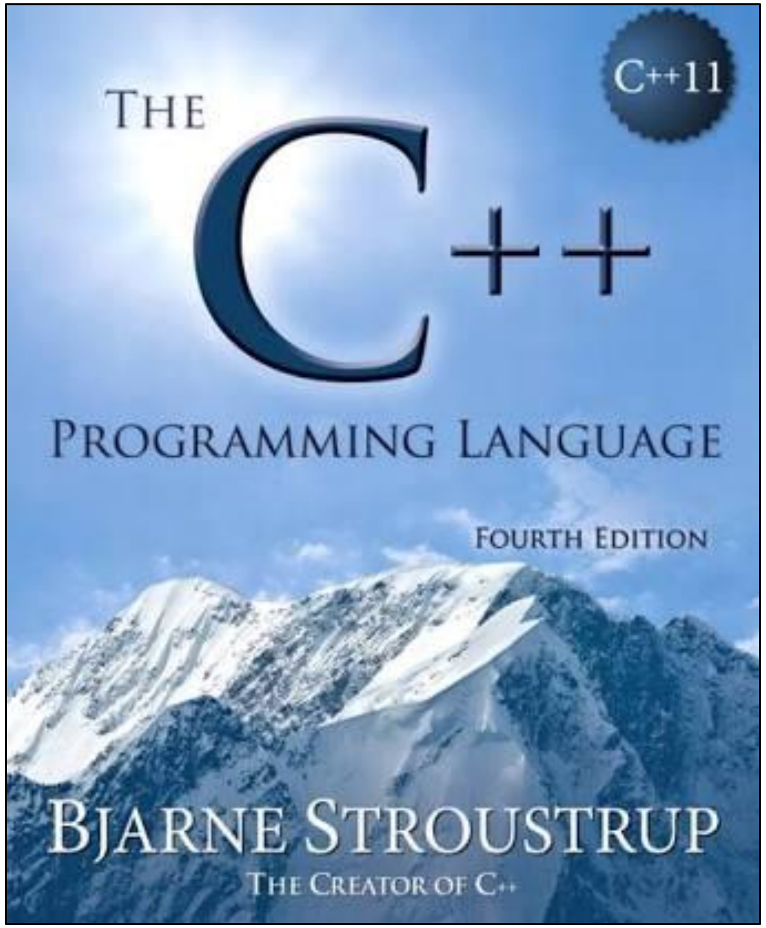
Interpreted Languages

- Slower at runtime
- Often faster to write program
- Farther away from the hardware
- No build phase

A good read on why is Python slow:
https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

References[1]





- Both C and C++ have excellent reference textbooks written by their creators
- TCPL terse, dense and readable – like C!
- TC++PL is exhaustive, detailed, and comprehensive – like C++!
- Audience poll – online tutorials you would recommend?

(1) Since this is a graduate course, we absolutely must have references!

Presentation Style and Topic Selection

- You can’t “learn” C++ in two hours.
- You can **get a sense** of what it’s all about.
- There are lots of great books and online tutorials.
- I’m going to show **complete programs** and let you **learn by example**
- I will show you **what I wish someone showed me** before AM 225
- I emphasize the **full toolchain** rather than just the “language”

Learning Goals

- **Get excited** about learning compiled languages!
- Quick tour of **what you can do** with C++
- Learn the **right search phrase** on Google or Stack Overflow for C++
- Share a repo of **working example programs** to refer to later
- Share a **working makefile** you can reuse as is for future projects
- Get to know each other a bit and **have some fun**

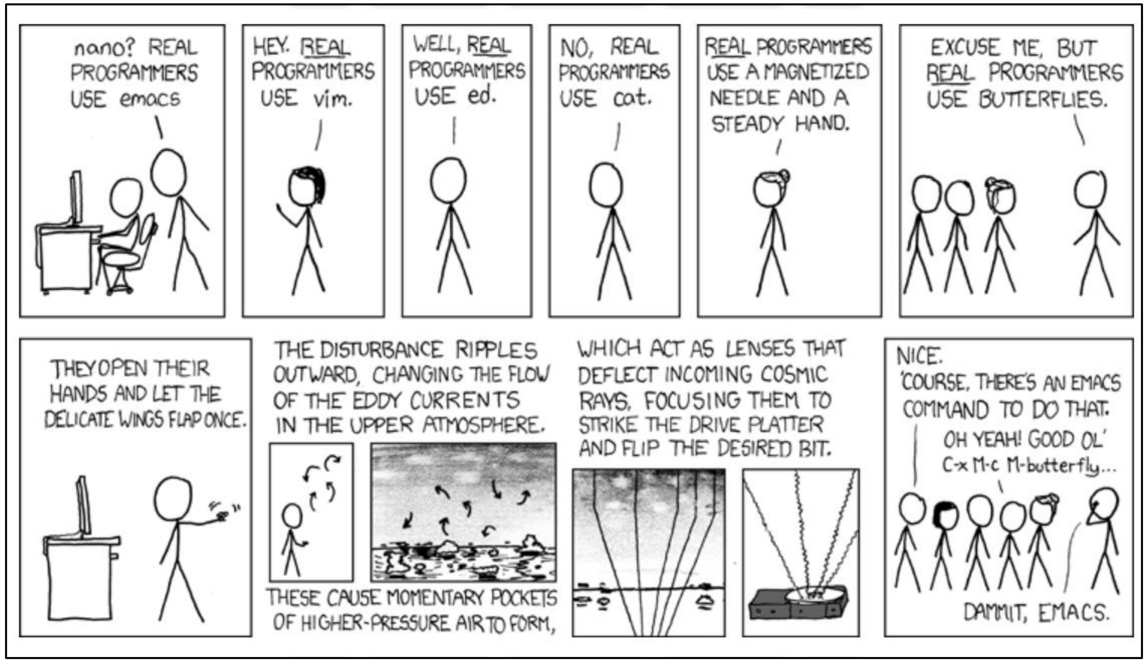
Hello World

Installing C++ on Your Computer

- You need to install a C++ compiler to compile and run a program
- A popular choice in the scientific community is gcc / g++
- Installation on Ubuntu 20.04 Linux:
  - `$sudo apt install gcc g++`
- Installation on Windows:
  - First install Windows Subsystem for Linux (WSL)
  - Install an Ubuntu 20.04 image, then follow previous steps
- Installation on Mac
  - Install XCode development tools from the App Store
  - Install MacPorts package manager
  - $ sudo port install gcc11
  - You can also use clang / clang++ on a Mac without MacPorts (just XCode)

Installing an IDE on Your Computer

- **IDE** = **I**ntegrated **D**evelopment **E**nvironment
- You can program C++ using only text editors
- But mere mortals are often more productive with an IDE
- I suggest **VS Code** – it's free, popular, and fully cross platform
  - It has great features to support both C++ and Python though add-ins
- Other popular choices include atom, sublime, Clion, notepad++, Xcode, nano, emacs, and vi(m)



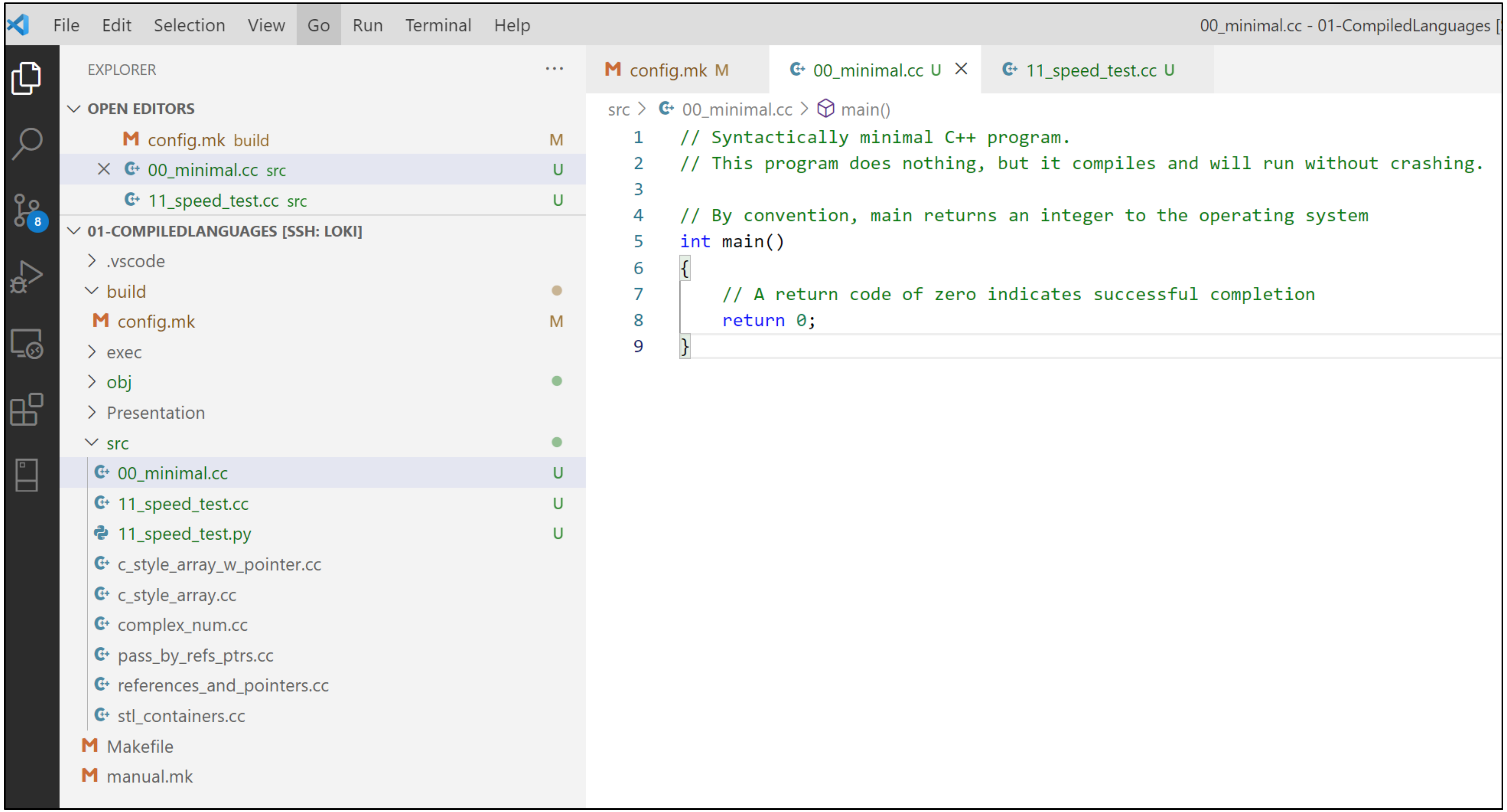
Cloning this Repository (git Crash Course)

- We don’t have time to cover version control and git today
  - Here is the minimum required to get through today’s activities
- Commands you can run to clone this repo to your computer:

```
$ cd ~/Harvard/AM-205
$ git clone https://github.com/chr1shr/am205_g_activities.git GroupAct --origin github
$ cd GroupAct/CompiledLanguages
```

- First we navigate to the parent directory of where we want to clone
- Then we ask git to create a copy of the repository on our local machine; this operation is called clone in git
- The URL is the location of a remote git repository hosted on GitHub
- Audience poll: please put suggestions about good online tutorials on Piazza

VS Code Screenshot of Minimal C++ Program



Hello World – MSE Style

src > 01_hello.cc > main()

```cpp
// Hello World C++ program.
// This program does slightly more than nothing.
// It demonstrates how to use the basic input / output facilities of C++.

// The #include directive tells the preprocessor to include the text of another C++ source file.
// This is always a "header file" in practice, with declarations (not definitions) of functions.
// This directive includes the declarations for the streaming input / output operations.
#include <iostream>

// The using directive brings the name cout into the namespace of our file
// cout is the name of the default output stream (text written to terminal in a console program).
using std::cout;

int main()
{
    // The traditional "hello world" program is supposed to write out
    // "Hello, world!"
    // However, I think that's a bit boring so I decided to quote a movie I liked.
    cout << "Hello. My name is Inigo Montoya.  You killed my father.  Prepare to die!\n";
    return 0;
}
```

Compiling and Running Hello World

```
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ ls
build  exec  Makefile  manual.mk  obj  Presentation  src
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ g++ src/01_hello.cc -o exec/01_hello.x
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ exec/01_hello.x
Hello. My name is Inigo Montoya.  You killed my father.  Prepare to die!
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
```

Activity 1: Hello World, Your Style

- Write a hello, world program with your favorite IDE
- Include a message that reflects **your personality**
- Compile and run your program in the terminal
- Feel free to refer to my example programs, but please don’t copy / paste any source code-we learn coding by typing!
- You might want to clone the example repo first; this is optional

C++ Data Structures (RPG Characters)

Built-In Data Types (Mostly Shared with C)

```
// Built-in Integral data types
bool b {true};                    // The bool data type is true or false
short s {0x7FFF};                 // Short integer; platform dependent, usually 16 bits
int i {0x7FFFFFFF};               // Default integer; platform dependent, usually 32 bits
long li {0x7FFFFFFFFFFFFFFF};     // Long integer; platform dependent, usually 64 bits
unsigned int ui {0xFFFFFFFF};     // Unsigned flavor of default integer; same size as int

// Integer types of fixed size
uint8_t ui8 {0xFF};               // 8-bit unsigned integer; use this for numbers, not char!
int16_t i16 {0x7FFF};             // Guaranteed to be 16 bits; signed
int32_t i32 {0x7FFFFFFF};         // Guaranteed to be 32 bits; signed
int64_t i64 {0x7FFFFFFFFFFFFFFF}; // Guaranteed to be 64 bits; signed
uint64_t ui64 {0xFFFFFFFFFFFFFFFF}; // Guaranteed to be 64 bits; unsigned

// Floating point data types
float f {pi};                     // The float data type is 32 bit IEEE single precision
double d {pi};                    // The double data type is 64 bit IEEE double precision
double err {f-d};                 // Rounding error between float and double

// Textual data types
char c {'a'};                     // The char data type is a 1 byte character
string str {"Hello"};             // Suggestion - use std::string rather than const char*

// Address of variables
int * ptri {&i};            // Pointer to integer; 32 or 64 bits based on architecture
```

Compile & Run data_types Program

```
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ g++ src/02_data_types.cc -o exec/02_data_types.x \
> -std=c++20 -Wall -Wextra -Wpedantic -Werror -O3 -lfmt
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ exec/02_data_types.x
Built-In Integral Types:
Type                  : Decimal               : Hex                   : Size
bool b                : true                  : (0x                1) : 1
short s               :                 32767 : (0x             7FFF) : 2
int i                 :            2147483647 : (0x         7FFFFFFF) : 4
long li               :   9223372036854775807 : (0x7FFFFFFFFFFFFFFF) : 8
unsigned int ui       :            4294967295 : (0x         FFFFFFFF) : 4
Integer Types of fixed size:
8 bit uint ui8        :                   255 : (0x               FF) : 1
16 bit int i16        :                 32767 : (0x             7FFF) : 2
32 bit int i32        :            2147483647 : (0x         7FFFFFFF) : 4
64 bit int i64        :   9223372036854775807 : (0x7FFFFFFFFFFFFFFF) : 8
64 bit unsigned ui64 : 18446744073709551615 : (0xFFFFFFFFFFFFFFFF) : 8

Floating Point Types:
float f    : 3.1415927410125732. 4 bytes.
double d   : 3.1415926535897931. 8 bytes.
double err : 0.0000000874227801.

Textual types:
char c     : a (97 as an int) (1 bytes).
string str : Hello (32 bytes).

Pointer type:
int * pi 0x7ffdd47fafcc. 8 bytes.
```

- Note the backslash to continue one long terminal command
- Note the C++ compilation arguments; set the language standard, warnings, and optimization level
- Note the –lfmt argument; asks the linker to link the format library fmt
- The actual library file on my system is /usr/lib/x86_64-linux-gnu/libformat.a
- g++ figures this out by searching the library path LD_LIBRARY_PATH

Structures and Enumerations (Shared with C)

```
// Enumeration for character alignments
enum class Alignment: int {good=1, neutral=2, evil=3};

// Structure for Star Wars character attributes
struct Character
{
    string name;
    int age;
    double force_ability;
    Alignment alignment;
};

// Declaration of function to print a character
void print_character(Character& c);
```

```
#include <string>
    using std::string;
#include <fmt/format.h>
    using fmt::print;
```

- Please use an enum instead of a "magic number" or string to encode an enumerated type!
- A struct definition has lines `type_name field_name;`
- Please remember the semicolon after the definition
- Note the ampersand in print_character declaration– more on this later!

A Closer Look at Printing the Output

```cpp
void print_character(Character& c)
{
    print("**********************************************
    print("Name: {:s}\n", c.name);
    print("Age: {:d}\n", c.age);
    print("Force Ability: {:3.1f}\n", c.force_ability);
    string align_str {""};
    switch (c.alignment)
    {
        case Alignment::good:
            align_str = "good";
            break;
        case Alignment::neutral:
            align_str = "neutral";
            break;
        case Alignment::evil:
            align_str = "evil";
            break;
    }
    print("Alignment: {:s}.\n", align_str);
}
```

```
$ exec/03_structures.x
***********************
Name: Luke Skywalker
Age: 22
Force Ability: 7.0
Alignment: good.
***********************
Name: Darth Vader
Age: 44
Force Ability: 8.0
Alignment: evil.
***********************
Name: Yoda
Age: 899
Force Ability: 9.0
Alignment: good.
```

Header Files and Using Directives

- `#include <string>` is an *include directive* to the preprocessor
- `using std::string;` is a *using declaration*
- We get pi with two statements:
  - `#include <numbers>`
  - `using std::numbers::pi;`
- We get a user-friendly print function with similar approach
  - `#include <fmt/format.h>`
  - `using fmt::print;`
- fmt::print implements the new C++20 format / printing standard
  - It provides a python-style print statement
  - It is type safe and more forgiving / easier to use than printf or `cout`

g++ Flags – Options & Libraries

- The g++ command included some options:
- -std=c++20 -Wall –Wextra –Wpedantic –Werror –O3
    - Use the C++20 language standard
    - Issue lots of warnings, and treat them as errors (good for learning)
    - Optimize to the highest level (we like fast programs)
- We need to install the fmt package and tell the linker about it!
    - $ sudo apt install libfmt-dev
- The installation puts the header files in /usr/include
- We need to tell g++ to link the fmt library with `-lfmt`

Activity 2: Silly Structures and Pretty Printing

- Create a silly struct to describe something that is amusing to you
  - I picked RPG-style character attributes for Star Wars characters
  - You can pick anything you like (feel free to use RPG for your favorite characters)
  - Please include one enum, one integer, one floating point type, and one string
- Then print out your information using the `fmt::print` function
  - You will need to install libfmt-dev and include the proper `-lfmt` flag for g++

C++ Functions (Square Root)

Square Root Function – Header and Definition

```cpp
// The declaration of sqrt_iter is in the header sqrt_iter.hh
double sqrt_iter(double x);
```

```cpp
// ******************************************************************************
// Calculate the square root of x iteratively
double sqrt_iter(double x)
{
    // If x is negative, just return not a number (NaN) instead of throwing an exception
    if (x<0) {return nan("");}
    // s is the current guess for square root of x; initialize with helper function
    double s {sqrt_guess(x)};
    // Set tolerance for convergence at machine epsilon in double precision
    double tol = 1.11E-16;
    // For at most 16 iterations, refine s iteratively
    // Compute dual lower and upper bounds, replace s with the midpoint
    for (int i=0; i<16; i++)
    {
        // New dual bound to s;
        double t = x / s;
        // Current relative error estimate
        double err = (s<t) ? (t-s)/s : (s-t)/t;
        // Improve estimate of s
        s = (s + t) / 2.0;
        // Break out of loop if tolerance achived
        if ( err < tol) {break;}
    }
    // By this point s should be a pretty good estimate for square root of x.
    return s;
}
```

The header file has a declaration. Always include the header file in the definition file.

Note the `if` statement.
Use braces to execute a block when (condition) is true.

Note the `for` statement.
`int i=0;` is the initializer
`i<16` is the loop test (continues when true)
`i++` is the loop update (at end)
`break` leaves the loop early

Note the ternary `?` operator:
(condition) ? value_if_true : value_if_false

The `return` statement sends the answer `s` back to the caller

Square Root Initial Guess

```cpp
// ************************************************************************
// Make a decent initial guess for the square root of x
double sqrt_guess(double x)
{
    // Decompose x into its mantissa and exponent in the form
    // x = m_x * 2^exp_x with m_x in {0} U [0.5, 1.0)
    int exp_x {0};
    double m_x {frexp(x, &exp_x)};
    // The exponent of the initial guess s is half the exponent of x
    int exp_s {exp_x / 2};
    // When exp_s is even, guess a mantissa of (1+m_x)/2
    // When exp_s is odd, guess a mantissa of (1+2m_x)/2 = 0.5 + m_x
    double m_s = (exp_x % 2) ? (1.0 + m_x)/2.0 : 0.5 + m_x;
    // initialize s to the double with mantissa m_s and exponent exp_s
    return ldexp(m_s, exp_s);
}
```

- The initial guess is often a critical part of iterative numerical algorithms
- This guess uses the representation of IEEE doubles to efficiently get close
- It will handle even very large or very small x
- The implementation details of `sqrt_guess` have been separated from the main iterative routine

Program to Use Square Root Function

```cpp
// Local dependencies
#include "sqrt_iter.hh"

// ******************************************************************************
// main like other C++ functions in most respects
// it is special in that program execution starts there
// it always has a return type of int.
// it can optionally accept arguments main(int argc, char* argv[]) from the command line
int main(int argc, char* argv[])
{
    // Should be 0 or 1 arguments
    if (argc>2)
    {
        print("Usage: sqrt [x].\n");
        print("Reports the square root of a double x passed at the command line.\n");
        print("If x is omitted, it defaults to 2.0.\n");
        exit(1);
    }

    // Populate x, the number whose square root we calculate
    double x = 2.0;
    if (argc == 2)
        {x = atof(argv[1]);}

    // Calculate sqrt(x) with library and hand rolled functions
    double s_lib {sqrt(x)};
    double s_mse {sqrt_iter(x)};
    // The error in the hand rolled function iterative
    double err = fabs(s_mse - s_lib);
```

- Note that the include is "sqrt_iter.hh" (in quotes) NOT <sqrt_iter>
- Library headers are included in angle brackets and by custom don't have .hh suffixes
- The compiler searches in different places depending on quotes vs. angle brackets
- Note that argc is the number of arguments including program name
- And argv is an array of C-style strings (const char* pointing to an array of characters, terminated by null)
- Library function atof converts a C-string to a double
- Library function fabs takes the absolute values – don't confuse this with abs!

Compile and Run Functions Program

```
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ g++ -c src/sqrt_iter.cc -o obj/sqrt_iter.o
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ g++ -c src/04_functions.cc -o obj/04_functions.o
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ g++ obj/04_functions.o obj/sqrt_iter.o -o exec/04_functions.x -lfmt
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ exec/04_functions.x
x            = 2.0000000000000000.
sqrt(x)      = 1.4142135623730951.
sqrt_iter(x) = 1.4142135623730949.
error        = 2.22e-16.
rel error    = 1.57e-16.
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages
$ exec/04_functions.x 3.0
x            = 3.0000000000000000.
sqrt(x)      = 1.7320508075688772.
sqrt_iter(x) = 1.7320508075688772.
error        = 0.00e+00.
rel error    = 0.00e+00.
```

- First we **compile** the two source files (sqrt_iter function and program itself) into object files
- Then we **link** the two object files into one executable file
- We can run without any arguments; x defaults to 2.0
- We can run with x = 3.0
- The program gets to within the machine epsilon on both
- I also verified that it works on very large numbers e.g. 2.0E12

Activity 3: Iterative Cube Root

- Write a function `cbrt_iter` to iteratively compute a cube root
  - You can use a similar approach to mine, where t = x / (s*s) is a dual bound
  - Or you can use any clever idea you come up with
- Write a program `test_cbrt.cc` to use your function and compare the results to the library function `cbrt` declared in `<cmath>`
- Separate the header file `cbrt_iter.hh` from the implementation `cbrt_iter.cc`
- Separately compile `cbrt_iter.cc` into `cbrt_iter.o` and `test_cbrt.cc` into `test_cbrt.o`
- Then link `test_cbrt.o` and `cbrt_iter.o` into `test_cbrt.x`
- Run your program and report the results

Pointers & Arrays (Matrix Multiplication)

Pointers and Operator new

```
// Initialize a random matrix and vector
init_t initialize(const int m, const int n)
{
    // Start the timer
    Timer t;

    // Allocate memory for arrays with operator new
    // The asterisk after double means that A, b and y are POINTERS to a double
    // Here, each of them is a pointer to an array of doubles, acontiguous block in memory
    double* A {new double[m*n]};
    double* x {new double[n]};
    double* y {new double[m]};

    // Initialize a random number generator
    rng_t rng {make_rng(42)};

    // Initialize the matrix A and vector x with uniform random numbers in [0, 1]
    random_matrix(A, m, n, rng);
    random_vector(x, n, rng);

    // Report initialization time
    print("********************************************************************************\n");
    print("Allocate and initialize a random {:d}x{:d} matrix, and a random {:d}x1 vector.\n", n, n, n);
    print("Time: {:5.4f} seconds.\n", t.tock());

    // Wrap allocated and initialized arrays into a structure
    return init_t {.A=A, .x=x, .y=y};
}
```

- The declaration `double* A` means that A is a **pointer** to double(s)
- That is, A holds the address in memory of a double. Here it is actually a whole array of doubles
- * is also the **deference operator**; so `*A` is a double (first element of the array)
- Operator new returns a block of memory; here it returns an array A of m*n doubles
- When you create an array with new, you must later call operator `delete`!
- This function returns three arrays A, x, y wrapped into a structure of type `init_t`.

Matrix Multiplication with Pointers and Arrays

```
// ****************************************************************************
// Matrix / vector multiplication function - hand rolled
// ****************************************************************************
// y is a pointer to the ANSWER - an mx1 column vector, y = Ax
// A is a pointer to an mxn matrix
// x is a pointer to an nx1 vector
// m is the number of rows in A and y
// n is the number of columns in A and rows in x
void matrix_mult_mse(double* y, const double* A, const double* x, int m, int n)
{
    // Iterate over the row number i
    for(int i=0; i<m; i++)
    {
        // Initialize sum in this row at zero
        y[i] = 0.0;
        // Accumulate the n entries in row i
        for(int j=0; j<n; j++)
        {
            // Offset for the (i,j) entry in A
            int k = n*i + j;
            // One term in the sum for y[i]
            y[i] += A[k] * x[j];
        }
    }
}
```

- The function arguments y, A, x are pointers to double
- Note that A and x are declared const
- but y is not
- This means the contents of A and x can't be changed, but those of y CAN change
- Whenever a pointer isn't written to, declare it const!
- I follow the convention of putting non-const pointer arguments first
- While the pointers themselves are "passed by value", their CONTENTS can be modified.
- The array index notation `A[k]` is equivalent to `*(A+k)`; it dereferences the double k to the right of A in memory

Matrix Multiplication “for real” with BLAS

```
// ***********************************************************************
// Matrix / vector multiplication function - using BLAS
// ***********************************************************************
void matrix_mult_blas(double* y, const double* A, const double* x, int m, int n)
{
    // Declaration of DEGEMV as ported to CBLAS; found in blas-netlib.h
    // void cblas_dgemv(CBLAS_LAYOUT layout,
    //                  CBLAS_TRANSPOSE TransA,
    //                  const int M, const int N,
    //                  const double alpha, const double *A, const int lda,
    //                  const double *X, const int incX, const double beta,
    //                  double *Y, const int incY);

    // Arguments for call to BLAS routine DGEMV
    CBLAS_LAYOUT layout {CblasRowMajor};
    CBLAS_TRANSPOSE TransA {CblasNoTrans};
    const  double alpha {1.0};
    const int lda {m};
    const int incX {1};
    const int incY {1};
    const double beta {0.0};
    // Delegate to DGEMV
    cblas_dgemv(layout, TransA, m, n, alpha, A, lda, x, incX, beta, y, incY);
}
```

- This is how I would multiply a matrix times a vector in a “real” C++ program
- I would choose not to reinvent the wheel, but to use a high quality library-BLAS
- BLAS is actually written in Fortran. The object file the linker uses is output by a Fortran compiler!
- In the header file I need to issue the directive `#include <cblas.h>`
- This is a “wrapper” that declares these functions so they can be called from a C or C++ program
- The function name DGEMV stands for **D**ouble precision **GE**neral **M**atrix **V**ector
- We need to tell BLAS some pretty low level information about the memory layout of the arrays
- This is annoying but helps BLAS to be fast.

Initializing Random Matrices

```
// ***********************************************************************
void random_matrix(double* A, int m, int n, rng_t& rng)
{
    // Initialize a uniform distribution between 0 and 1 using the RNG
    std::uniform_real_distribution<double> unif {make_uniform_dist()};

    // Iterate over the row, i, going up to m
    for(int i=0; i<m; i++)
    {
        // Iterate over the column, j, going up to n
        for(int j=0; j<n; j++)
        {
            // Offset for the (i, j) matrix entry
            int k = n*i + j;
            // Populate A[i, j] with a random number in [0, 1)
            A[k] = unif(rng);
        }
    }
}
```

- Note the function parameter `rng_t& rng`
- This means the random number generator object rng is passed by reference
- Equivalent to passing a pointer `rng_t* prng`, and every appearance of `rng` is replaced by `(*prng)`
- Good rule of thumb: when passing a large object as a function parameter, pass it by **constant reference**, e.g. `const T& arg`
- In this case, I pass rng by non-const reference because I WANT the rng to be modified so the vector doesn't sample the same values

Matrix Multiplication Program in C++

```cpp
int main()
{
    // The size of the matrix A (number of rows and columns)
    constexpr int m {10000};
    constexpr int n {10000};
    // The number of trials
    constexpr int num_trials {100};

    // Initialize the matrix A and the vectors x, y
    init_t mats {initialize(m, n)};
    // Unpack the arrays A, x, y from mats
    double* A {mats.A};
    double* x {mats.x};
    double* y {mats.y};

    // Test the matrix_mult routine (hand rolled)
    test_matrix_mult(matrix_mult_mse, "matrix_mult_MSE", A, x, y, m, n, num_trials);

    // Test the matrix_mult routine (BLAS)
    test_matrix_mult(matrix_mult_blas, "matrix_mult_BLAS", A, x, y, m, n, num_trials);

    // Delete manually allocated arrays
    delete [] A;
    delete [] x;
    delete [] y;
}
```

- I declared the integers m and n `constexpr` because they are known at compile time
- The function `test_matrix_mult` takes a function pointer to a matrix multiplication routine
- This allows me to avoid repeating similar code for testing the hand rolled and BLAS versions
- I manually delete the arrays with `operator delete []`
- The **golden rule** of operator new is "*delete [] unto arrays as ye have created them with operator new*"

Speed of Matrix Multiplication in C++

```
michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages 2021-08-31 14:53:41
$ exec/05_pointers.x
********************************************************************************
Allocate and initialize a random 10000x10000 matrix, and a random 10000x1 vector.
Time: 1.5421 seconds.
********************************************************************************
Matrix / vector product of size 10000, repeated 100 times. Using function matrix_mult_MSE.
Mean entry of y=Ax: 2520.134.
Time: 10.723 seconds.
Mean Time:  107.232 milliseconds per trial.
********************************************************************************
Matrix / vector product of size 10000, repeated 100 times. Using function matrix_mult_BLAS.
Mean entry of y=Ax: 2520.134.
Time: 2.483 seconds.
Mean Time:   24.830 milliseconds per trial.
```

- The obvious implementation of matrix multiply with nested for loops costs 107.2 ms per multiply
- Delegating to the heavily optimized DGEMV costs 24.8 ms per multiply
- That is not bad- a mindless code is within a factor of 4.3 of “best possible” performance

Matrix Multiplication in Python

```
# ***************************************************************
def matrix_mult(A: np.ndarray, x: np.ndarray) -> np.ndarray:
    """Multiply matrix A by vector x by hand in Python"""
    # Extract size m x n from A
    m, n = A.shape
    # Initialize an array for the answer y
    y: np.ndarray = np.zeros(m)
    # Iterate over rows i
    for i in range(m):
        # Iterate over rows j
        for j in range(n):
            y[i] += A[i,j] * x[j]
    # Return the product vector, y
    return y
```

- This is the closest Python analog I could write to the C++ `matrix_mult_mse` function
- It uses numpy arrays for storage of the input and output

Speed of Matrix Multiplication in Python

```
(am205) michael@Loki: ~/Harvard/AM-205/AM-205-MSE/03-GroupAct/01-CompiledLanguages 202
$ python src/06_matrix_mult.py
******************************************************************************
Allocate and initialize a random 10000x10000 matrix, and a random 10000x1 vector.
Time: 0.8584 seconds.
******************************************************************************
Matrix / vector product of size 10000x10000, repeated 100 times using numpy.dot.
Time: 2.4935 seconds.
Mean Time: 24.9354 milliseconds per trial.
Mean entry of y=Ax: 2509.569.
******************************************************************************
Matrix / vector product of size 10000x10000, repeated 1 times using matrix_mult.
Time: 38.453 seconds.
Mean Time: 38452.991 milliseconds per trial.
Mean entry of y=Ax: 2509.569.
******************************************************************************
Matrix / vector product of size 10000x10000, repeated 1 times using matrix_mult_list.
Time: 10.049 seconds.
Mean Time: 10049.439 milliseconds per trial.
Mean entry of y=Ax: 2509.569.
```

- numpy.dot is a high quality library function written in C
- I think it is delegating to DGEMM
- Therefore it is fast! 24.9 milliseconds per multiply
- This is even faster than our naïve C++ implementation
- But an apples-to-apples comparison is a function written in pure Python
- That is **much slower**, costing **38453** or **10049** ms per multiply
- That is **slower than** the **C++** implementation by a **factor of 398 or 104**

Activity 4: Symmetric Matrix Multiplication

- One of the most important special cases of matrices are symmetric:
- A symmetric matrix **A** satisfies $a_{ji} = a_{ij}$
- Write a function `mv_mult_sym` that multiplies a symmetric matrix A by an arbitrary vector x
  - The input array A should contain n(n+1)/2 doubles, NOT n*n of them
  - You might want a separate function to compute array offset k given row i and column j; I like the function name `ij2k`
  - The BLAS subroutine for this operation is `DSYMV`

Classes (Timer)

Short Example: Class Declaration

```
class Timer
{
public:
    /// Constructor
    Timer();

    /// Set time time point
    void tick();

    /// Return the elapsed time in seconds without any string
    double tock();

    /// Take a "split" by calculating elapsed time, then re-starting timer
    double split();

    /// Return the elapsed time in seconds
    double tock_msg(const string blurb = "");

private:
    /// Static time point that is updated each time tick() is called
    highResTimePoint tp0;
    /// Static time point that is updated each time tock() is called
    highResTimePoint tp1;
};
```

- This is the declaration of a real class I wrote named `Timer`
- It's used to time operations and is modeled after the `tic` and `toc` functions in Matlab
- The *constructor* is a special method that builds an instance of the class
- This class doesn't need a *destructor* (the default one is OK)
- It has *methods* in its public interface (API) called `tick`, `tock`, `split`, and `tock_msg`
- It also has two private data *members*, `tp0` and `tp1`, that are time points
- These are private because they're implementation details the public consumer shouldn't see

Short Example: Class Definition

```
// ************************************
// Constructor
Timer::Timer() :
    tp0 {highResTimePoint()},
    tp1 {highResTimePoint()}
{
    tick();
}

// ************************************
// Start the timer.
void Timer::tick()
    {tp0 = high_resolution_clock::now();}
```

```
// ****************************************************
double Timer::tock()
{
    // Time point when tock() is called
    highResTimePoint tp1 = high_resolution_clock::now();

    // The elapsed time in nanoseconds
    time_unit_t t = duration_cast<nanoseconds>(tp1 - tp0).count();

    // Compute the elapsed time in seconds.
    double tSeconds = static_cast<double>(t) / aBillion;

    // Return the elapsed time in seconds
    return tSeconds;
}
```

- This is an excerpt of the implementation of the Timer class
- The constructor uses a special syntax with the colon called *member initialization*
- All the data elements (here the two highResTimePoint objects) are initialized
- The methods are defined in the `namespace` of the class (a class is a namespace)
- Any C++ file that uses class Timer should `#include “Timer.hh”`.
- This file `Timer.cc` will be compiled into `Timer.o`, which should be passed to the linker

Build Automation with GNU Make

Build Automation with GNU make

```
# ****************************************************************
# Configuration options
# ****************************************************************

# Make settings: warn on unset variables, use parallel processing
# MAKEFLAGS+=--warn-undefined-variables -j
MAKEFLAGS+=-j

# Directory layout
SRC_DIR := src
OBJ_DIR := obj
EXE_DIR := exec

# C++ compiler
CXX := g++

# C++ Compilation flags
CXX_FLAGS := -std=c++20 -Wall -Wextra -Wpedantic -Werror -O3
# Selected libraries
LD_DIRS :=
LD_LIBS := -lfmt -lblas -lgmp

# Combined LD_FLAGS arguments to linker - library search path and libraries
LD_FLAGS := $(strip $(LD_DIRS) $(LD_LIBS))
```

- This is the first part of a working makefile for the programs presented today
- All variables in a makefile are strings!
- If you name this file “Makefile” or “makefile”, you can run it by typing `$make` at the commandline
- You can run make on any makefile you like via e.g.
  `$make -f my_makefile.mk`
- The only thing this part of the file does is set up some string variables
- I name three directories of interest, for source, object and executable files
- I also choose the C++ compiler, set the C++ compilation options, and set the linker options.
- The := assignment evaluates right away; use this instead of = unless you need recursive expansion

Collections of Files & Targets in a makefile

```
# *******************************************************************************
# Collections of files and targets
# *******************************************************************************

# All the executable targets (stem only)
TGT_EXE := \
    00_minimal 01_hello 02_data_types 03_structures 04_functions 05_pointers \
    11_c_array_auto 12_c_array_new 13_complex 14_containers 15_refs_pointers 16_pass_by_ref_ptr \
    21_pi
# $(info TGT_EXE = $(TGT_EXE))

# Executable program files - build from targets list
EXECS := $(patsubst %, $(EXE_DIR)/%.x, $(TGT_EXE))
# $(info EXECS = $(EXECS))

# All the source files (full filename) using wildcard function
SRC_ALL := $(wildcard $(SRC_DIR)/*.cc)
# $(info SRC_ALL = $(SRC_ALL))

# All the targets (stem only)
TGT_ALL := $(patsubst src/%.cc, %, $(SRC_ALL))
# $(info TGT_ALL = $(TGT_ALL))

# All of the object files; written out as full file names with path e.g. obj/my_file.o
OBJ_ALL := $(patsubst %, obj/%.o, $(TGT_ALL))
# $(info OBJ_ALL = $(OBJ_ALL))
```

- This part of the makefile assembles collections of files and targets used later
- A collection is just a long string of tokens separated by whitespace
- You can think of TGT_EXE as loosely meaning to GNU make the same thing `['00_minimal', '01_hello', ... '21_pi']` would mean in a Python program
- SRC_ALL and TGT_ALL use the string substitution function `$(patsubst )`
- `SRC_ALL = src/00_minimal.cc` …
- `OBJ_ALL = obj/00_minimal.o` …
- You can use `$(info )` to print these out to the console during development

Recipes and Rules: Manual

```
# ************************************************************************
# Completely manual recipes to build two program (teaching purposes only!)
# ************************************************************************


# Build 00_minimal executable program
exec/00_minimal.x: src/00_minimal.cc
	$(CXX) src/00_minimal.cc -o exec/00_minimal.x \
	$(TAB) $(CXX_FLAGS) $(INCLUDE) $(CXX_MACROS) $(LD_FLAGS)

# Build 01_hello executable program
exec/01_hello.x: src/01_hello.cc
	$(CXX) src/01_hello.cc -o exec/01_hello.x \
	$(TAB) $(CXX_FLAGS) $(INCLUDE) $(CXX_MACROS) $(LD_FLAGS)
```

- The first **rule** says that the *target* exec/00_minimal.x *depends* on src/00_minimal.cc
- If the timestamp on the executable is older than the source file, make will rebuild it…
- … by running the **recipe** in the rule text e.g., `g++ src/00_minimal.cc -o exec/00_minimal.x -std=c++20 -lfmt`
- This rule text is passed to the shell
- This approach is already very powerful compared typing in commands by hand, but involves a lot of repetitive recipes and rules

Recipes and Rules with Static Patterns

```
# ************************************************************************
# Manually list extra dependencies for executables that have them in rule specific variables
# ************************************************************************
exec/04_functions.x : LINK_OBJ = obj/sqrt_iter.o
exec/05_pointers.x  : LINK_OBJ = obj/matrix_mult.o obj/Timer.o
```

```
# ************************************************************************
# Static pattern rules
# ************************************************************************
# Static pattern rule to compile source files into object files
$(OBJ_ALL) : $(OBJ_DIR)/%.o : $(SRC_DIR)/%.cc
	$(CXX) -c $< -o $@ $(CXX_FLAGS) $(INCLUDE) $(CXX_MACROS)

# Define the command to link an executable from its dependent object files
define CXX_LINK
	$(CXX) -o $@ $^ $(LD_FLAGS)
endef

# Static pattern rule to link each executable from its corresponding object file
.SECONDEXPANSION:
$(EXECS) : $(EXE_DIR)/%.x : $(OBJ_DIR)/%.o $$(LINK_OBJ)
	$(CXX_LINK)

# Static pattern rule that each target name depends on its sister executable file
$(TGT_ALL) : % : $(EXE_DIR)/%.x
```

- This snippet shows how to automate the rules that we saw manually
- The first rule can be interpreted as follows:
- For every object in the variable OBJ_ALL (`obj/00_minimal.o` …)
- There is a rule of the form
  `obj/[target].o : src/[target].cc`
- The recipe for this rule is to compile the course file into the object file
- There as also a family rules for each executable target.
- For each target name e.g. 00_minimal, there is a rule of the form

`exec/[target].x: obj/[target].o [extra objects]`

- This rule is to link the object file(s) into the executable
- The last rule says that the target `00_minimal` depends on the file `exec/00_minimal.x`

“Convenience” Targets all, clean

```
# ****************************************************************************
# Convenience targets
# ****************************************************************************

# Make conventional target "all" depend on all the executables
all: $(EXECS)

# Set phony targets
.PHONY: all clean

# Set the default goal
.DEFAULT_GOAL: all

# Target clean removes all the built executable and object files.
# Use wildcards to ensure any "orphaned" object files or executables are also deleted.
clean:
	@rm -f $(OBJ_DIR)/*.o $(EXE_DIR)/*.x
```

- The targets we’ve seen so far are files we can build from other files
- But some targets aren’t files, they’re tasks we want to accomplish
- The two most common such “convenience targets” are to build everything and delete everything
- By convention, the rules to do these two task are named `all` and `clean`
- We tell make that these aren’t files by making them depend on .PHONY
- We tell make that the default goal is all by making all depend on the special target .DEFAULT_GOAL
- If we just type `$make` it will build all

Advice about GNU Make and Makefiles

- GNU Make and makefiles can seem very cryptic at first
- Don’t be afraid to copy / paste an example Makefile and adapt it
- A major learning goal of this section was to give you a template makefile you can use for future C / C++ projects
- The repository has a more sophisticated version that includes automatic dependency generation