INTRODUCTION TO PROGRAMMING IN C++

COMPILERS VERSUS INTERPRETERS

- A **compiler** produces a stand-alone binary for a given *platform* (cpu+operating system). The output of a compiler is an *object file*, represented with a `.o` suffix on Unix.
- A **linker** takes the `.o` files and any external *libraries* and links them into the executable. Normally the linker is invoked through the compiler.
- An **interpreter** interprets line by line. The binary that is run is the interpreter itself. Programs for interpreters are often called *scripts*. Scripts are frequently cross platform, but the interpreter itself must be appropriate to the platform.

COMPILED LANGUAGES

- Compiled languages are:
  - Generally stricter about typing (static typing) and memory allocation.
  - Generally produce faster and more efficient runs.
- Interpreted languages are:
  - Generally looser about typing (dynamic typing).
  - Generally have dynamically sized data structures built in.
  - Often run very slowly.

STRENGTHS AND WEAKNESSES

C++ (not C)

- Limited mathematical built-ins
- True multidimensional arrays not possible without add-on libraries (Blitz++, Boost)
- Pretty good string handling (compared to C)
- Straightforward implementation of classes (but no modules)

Fortran

- (2003/8) Many math function built-ins
- Multidimensional arrays a first-class data structure, array operations supported
- Does not support true strings yet, just character arrays
- Classes somewhat clunky. Modules fill much of this role.

SETTING UP YOUR ENVIRONMENT

INTEGRATED DEVELOPMENT ENVIRONMENTS

- An Integrated Development Environment (IDE) combines an editor and a way to compile and run programs in the environment.
- A well-known IDE for Microsoft Windows is VisualStudio. Available through Microsoft Store, not free for individuals.
- Mac OSX uses Xcode as its native IDE. Xcode includes some compilers, particularly for Swift, but it can manage several other languages. Available at App Store, free.
- A full-featured cross-platform IDE is Eclipse (www.eclipse.org). Free.
- A lighter-weight IDE for Windows and Linux is Code::Blocks (www.codeblocks.org). Free.
- We will use a very lightweight IDE called Geany since it is free, easy to install and use, and works on all three platforms.

LINUX

- For users of the University of Virginia's cluster, first load a compiler module.

```
module load gcc
```

brings a newer gcc, g++, and gfortran into the current environment

```
module load geany
```

```
geany &
```

- Geany is also available for all popular Linux distributions and can be installed through the distribution's package manager.

WINDOWS AND MAC

- Geany can be installed on Windows and Mac through the usual software installation methods.
- Geany does not install a compiler suite. This must be performed independently.
- Macs with Xcode include gcc and g++ but not gfortran.
- Windows does not include a default compiler suite, but VisualStudio includes Microsoft C and C++.
- Geany can be downloaded for Mac or Windows starting from its home page
  - www.geany.org

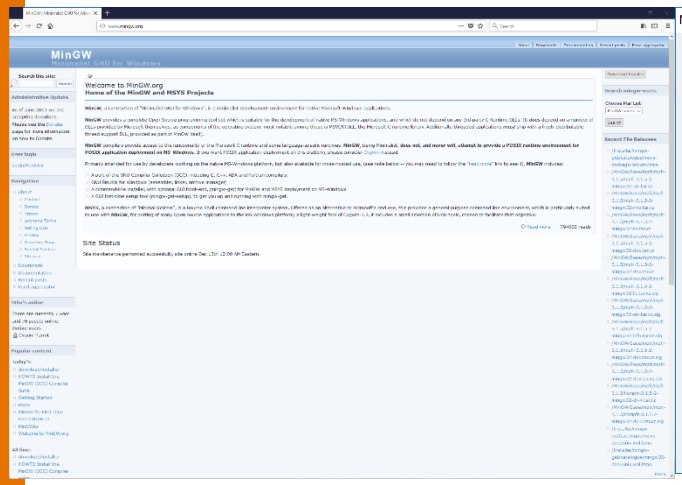
INSTALLING COMPILERS ON MACS

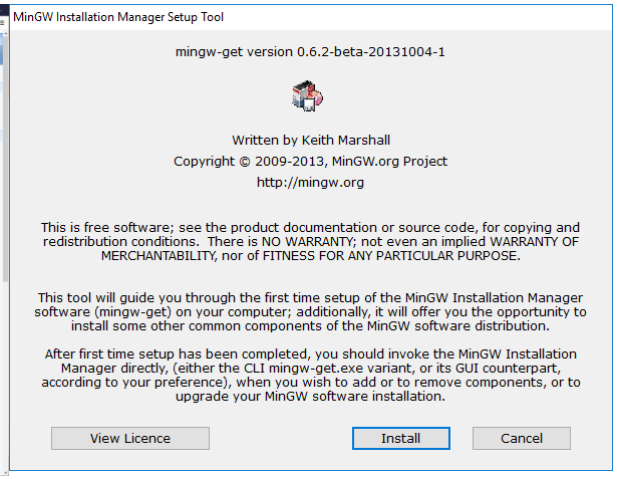
- Install Xcode from the App Store.
- If you are going to use Fortran, download a binary for your version of OSX from
- https://gcc.gnu.org/wiki/GFortranBinaries

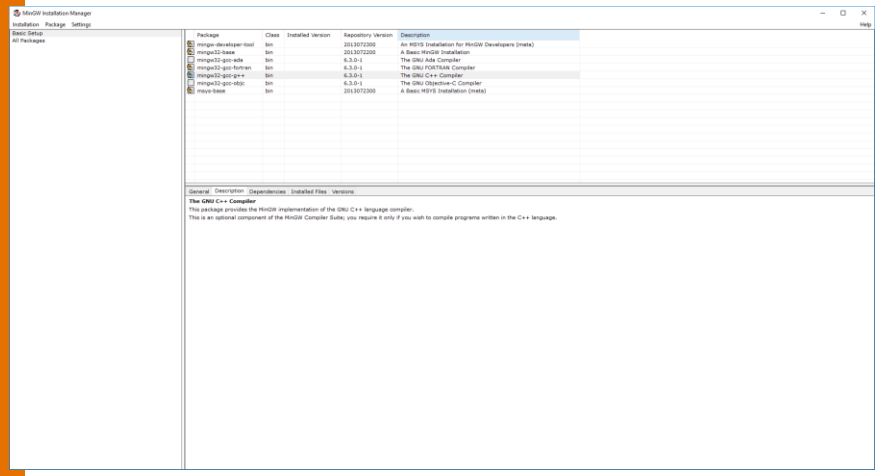
INSTALLING COMPILERS ON WINDOWS

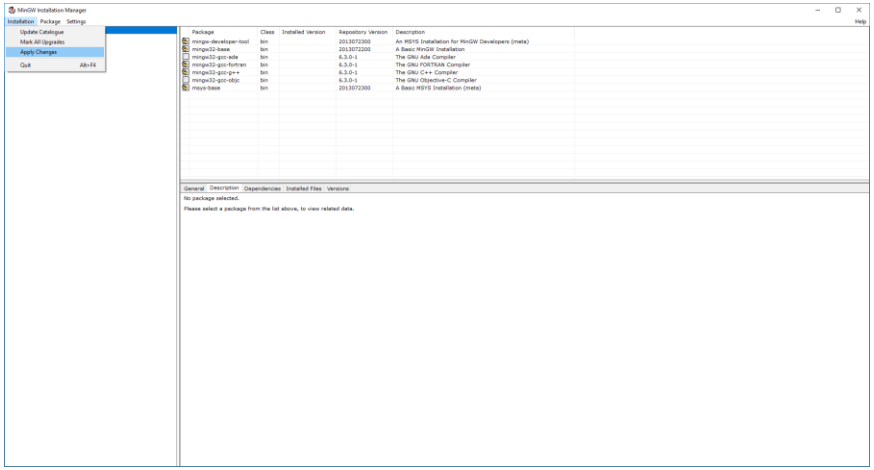
- MinGW provides a free distribution of gcc/g++/gfortran
- Executables produced by the standard MinGW package will be 32 bits
- Also install MSYS for a minimalist Unix system.
  - Download from www.mingw.org
  - Run installer
  - Choose packages to install, then click Apply.
  - After the installation, follow instructions for "After Installing" at http://www.mingw.org/wiki/Getting_Started
  - Be sure to modify your path environment variable

MINGW


MinGW
Welcome to MinGW.org
Home of the MinGW and MSYS Projects
Site Status


MinGW Installation Manager Setup Tool
mingw-get version 0.6.2-beta-20131004-1
Written by Keith Marshall
Copyright © 2009-2013, MinGW.org Project
http://mingw.org
This is free software; see the product documentation or source code, for copying and redistribution conditions. There is NO WARRANTY; not even an implied WARRANTY OF MERCHANTABILITY, nor of FITNESS FOR ANY PARTICULAR PURPOSE.
This tool will guide you through the first time setup of the MinGW Installation Manager software (mingw-get) on your computer; additionally, it will offer you the opportunity to install some other common components of the MinGW software distribution.
After first time setup has been completed, you should invoke the MinGW Installation Manager directly, (either the CLI mingw-get.exe variant, or its GUI counterpart, according to your preference), when you wish to add or to remove components, or to upgrade your MinGW software installation.
View Licence
Install
Cancel

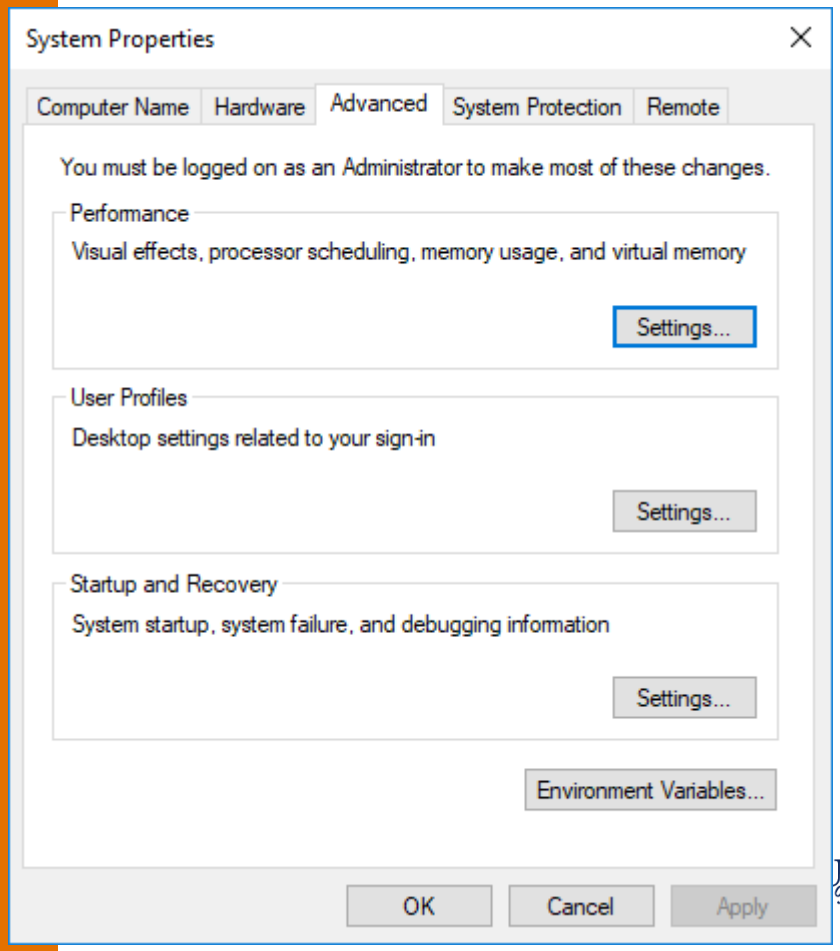

MinGW Installation Manager

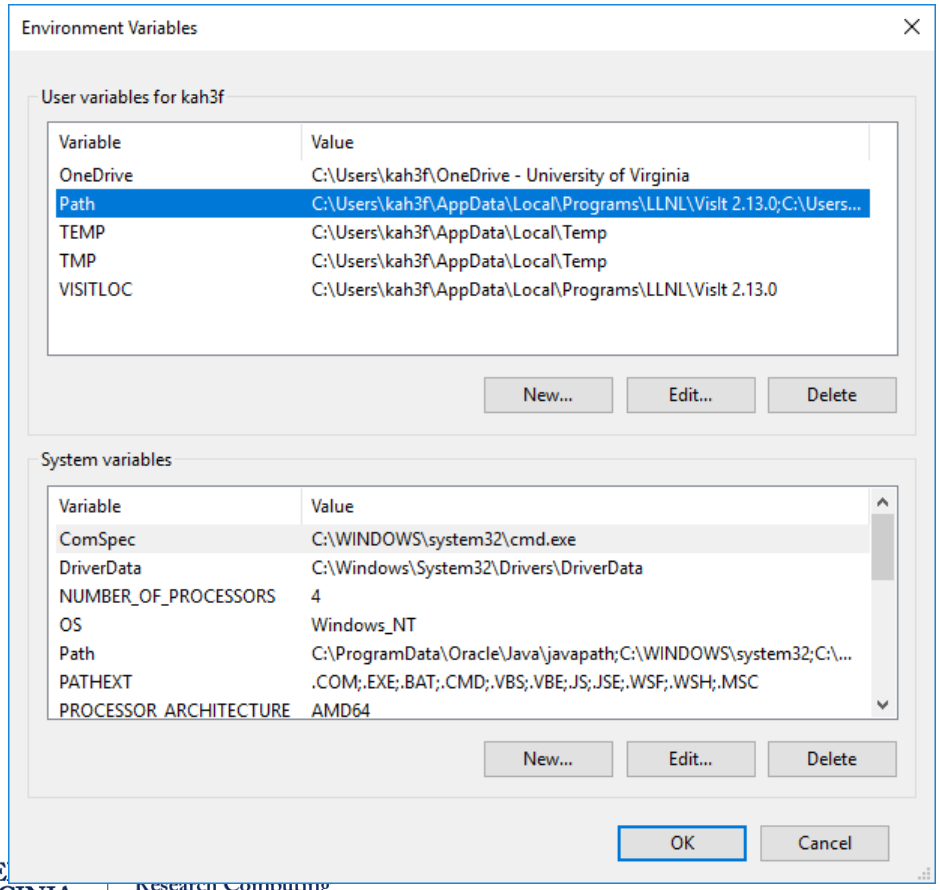

MinGW Installation Manager
Update Catalogue
Mark All Upgrades
Apply Changes
Quit

ENVIRONMENT VARIABLES IN WINDOWS

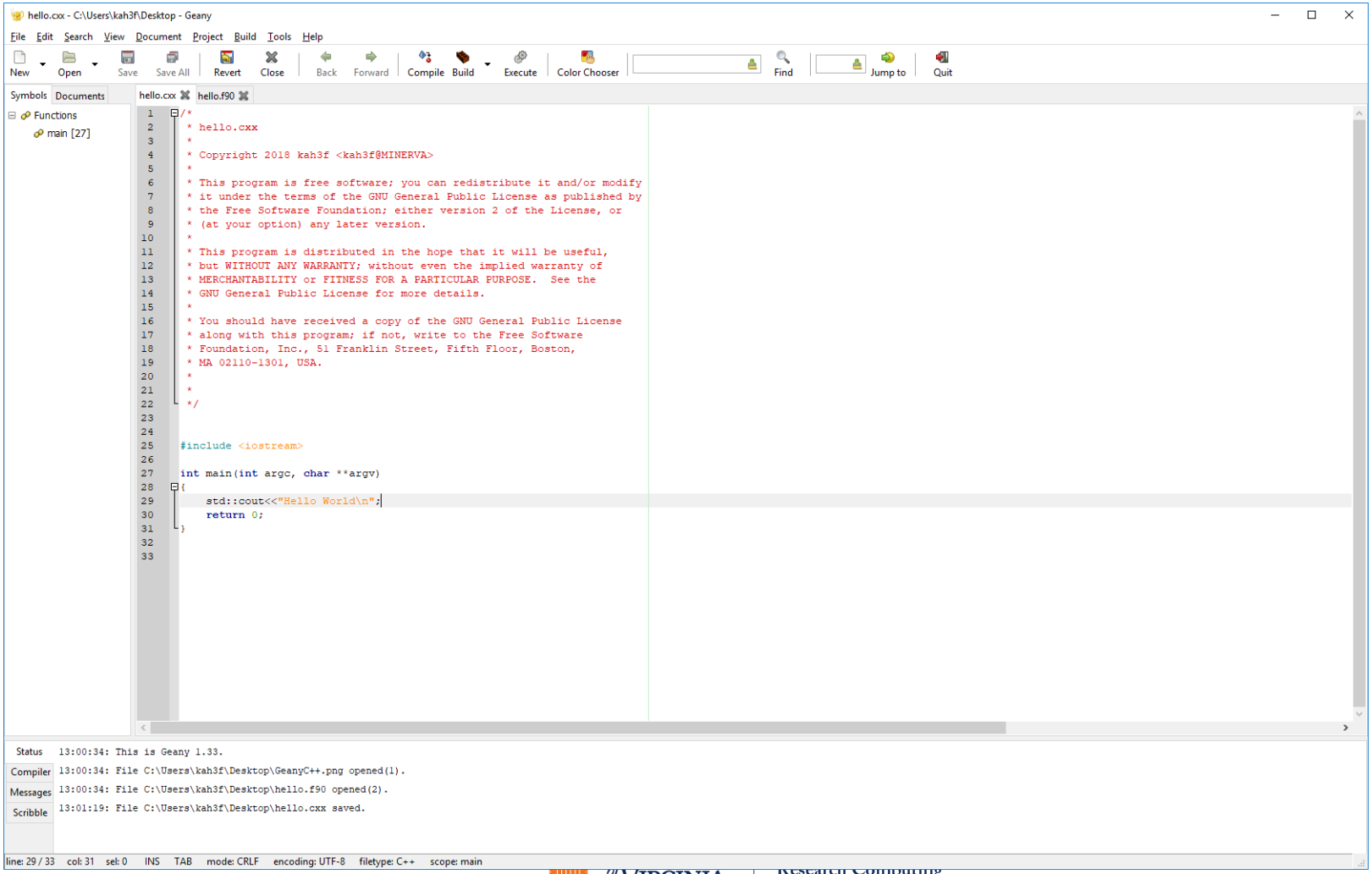
Control Panel->System and Security->Advanced system settings->Environment Variables

Once you open Path, click New to add to the Path





GEANY ON WINDOWS



VARIABLES

VARIABLES IN C++

- Like most programming languages, C++ is case sensitive. Variables `Mean` and `mean` are different to the compiler.
- Like most compiled languages, C++ is *statically typed*. All variables must be *declared* to be of a specific type before they can be used. A variable's type cannot be changed once it is declared.
- C++ is (nearly) strongly typed. Mixed-mode expressions are limited and most conversions must be explicit.

NUMERIC TYPES: INTEGER

- Integer (int)
  - Quantities with no fractional part
  - Represented by sign bit + value in *binary*
    - *Remember that computers do not use base 10 internally*
    - Default integers are of size 32 bits
  - Maximum signed integer is is 2^{32}-1
    - `unsigned int` is a type that covers only nonnegative ints
  - Other types
    - `long` may be either 32 or 64 bits depending on compiler/platform
      - Standard requires only that it be at least 32 bits.
      - Usually 32 bits on Windows, 64 bits on other platforms now.
    - `short` is 16 bits
    - `long long` is a 64-bit integer (C++11 standard, before that an extension)

NUMERIC TYPES: SINGLE PRECISION

- Floating point single precision `float`
  - Sign, exponent, mantissa
  - 32 bits
  - IEEE 754 defines representation and operations
  - Approximately 7-8 decimal digits of precision, *approximate* exponent range is 10^{-126} to 10^{127}

NUMERIC TYPES: DOUBLE PRECISION

- Double precision floating point `double`
  - Sign, exponent, mantissa
  - 64 bits
    - Number of bits NOT a function of the OS type! It is specified by the IEEE 754 standard!
  - Approximately 15-17 decimal digits of precision, approximate exponential range 10^{-308} to 10^{308}

NON-NUMERIC TYPES: BOOLEAN

- Booleans are represent truth value `bool`
- Values can be `true` or `false`
- Internally `true` is `1` and `false` is `0`, but it's easier for humans to read and remember true/false.

NON-NUMERIC TYPES: CHARACTER

- Character char
  - 1 byte (8 bits) per single character
- A character has a fixed length that must be declared at compile time, unless it is treated as allocatable (more on that later).

```
char[8] mychar;
```

- The default length is 1, however.

```
char letter;
```

NON-NUMERIC TYPES: STRING

- A string is a sequence of characters of variable length.
- Requires adding a header
- `#define <string.h>`
- The string is a *class*, which is a little beyond our scope right now. But you can still use basic functions without understanding the class.

```
string str, str1, str2;
str.size();  // length of string
str1+str2; // concatenate two strings
str.substr(2,5); // substring (counts from 0)
```

LITERALS

- Literals aka constants
  - Specified values e.g.

```
3
3.2
3.213e0
"This is a string"
"Isn't it true?"
true
```

Literals have a type but it is determined from the format rather than a declaration.

VARIABLE DECLARATIONS

- Variables are declared by indicating the type followed by a comma-separated list of variables followed by a semicolon.

```
int i,j,k;
float x, y;
```

INITIALIZING AT COMPILE TIME

- Variables can be declared and initialized at the same time:

```
float x=1.e-8, y=42.;
int i,j,k,counter=0;
```

POINTERS AND REFERENCES

- A pointer is a variable that points to a location in memory.
  - C++ has ways to avoid pointers, but they still appear regularly in much code.
- Pointers are declared with *

```
float *x, y;
```

  - x is a pointer, y is a variable.
- The value of a pointer is obtained explicitly by the *dereference operator* &

```
y=99.;
x=&y; //x now points to location of y
cout<<x<<" "<<*x<<" "<<y<<"\n";
0x7ffe577a0494 99 99
```

EXAMPLE

- Start Geany (or whatever editor you want to use). Type

```
#include <iostream>
int main() {
/* My first program
   Author:  Your Name
*/
  float x,y;
  int i,j=11;
  x=1.0;
  y=2.0;
  i=j+2;
  std::cout<<"Reals are "<<x<<" "<<y<<"\n";
  std::cout<<"Integers are "<<i<<" "<<j<<"\n";
  return 0;
}
```

CONST

- In compiled languages, programmers can declare a variable to have a fixed value that cannot be changed.
- In C/C++ this is indicated by the `const` attribute.

```
const float pi=3.14159;
```

- Attempting to change the value of a variable declared to be a parameter will result in a fatal compiler error.

TYPE CONVERSIONS

- Most compilers will automatically cast numeric variables to make mixed expressions consistent. The variables are promoted according to their rank. Lowest to highest the types are integer, float, double, complex.
- Use explicit casting to be clear, or in circumstances such as argument lists where the compiler will not do it.
- Strings may be cast to numbers and vice versa by a stringstream (this is the "correct" C++ way).

EXAMPLES

- Explicit casting among numeric types, default kind.

```
R=(float) I;
I=(int) R;
D=(double)R;
```

CHARACTER ⇔ NUMERIC

- Stringstreams are internal string *buffers*.
- Convert numeric to character:

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main() {
   string age;
   int iage;
   iage=39
   stringstream ss;
   ss<<iage;        //load iage into buffer
   age=ss.str();
```

- Convert character to numeric

```
   age='51'
   stringstream ss2(age);
   ss2>>iage;
}
```

ARITHMETIC OPERATIONS

- Operators are defined on integers, floats, and doubles
- `+ -` add subtract
- `* /` multiply divide
- Operator Precedence is:
- `(* /) (+ -)`
- Evaluation is left to right by precedence unless told otherwise with parentheses

INTEGER OPERATORS

- In C++ 2/3 is always zero! Why?
  - Because 2 and 3 are both integers. Nothing will be promoted to a float, so / is an integer operation that yields an integer result
- Remainder can be obtained from %
  - Use for negatives is uncommon in all languages and may not behave as you expect.

BOOLEAN OPERATORS

- Negation
  - `!`

    ```
    !flag
    ```
- AND
  - `&&`
- OR
  - `||`

CONDITIONAL OPERATORS

- Numeric
  - equals ==
  - not equal /=
  - strictly less than <
  - strictly greater than >
  - less than or equal to <=
  - greater than or equal to >=

CONDITIONAL OPERATOR PRECEDENCE

- `>`, `>=`, `<`, `<=` outrank `==` or `!=`
- `==`, `!=` outranks `&&`
- `&&` outranks `||`
- As always, use parentheses to change grouping or to improve clarity.

EXERCISE

Exercises with conditionals.

Be sure to declare variables appropriately.

```
a=11.; b=9.; c=45; n=3;
cout<< boolalpha << (a>b)<<"\n";
cout << (a<b && c==n) << "\n";
cout << (a<b || c==n) << "\n";
cout << (a>b || c==n && a<b) <<"\n";
cout <<(a>b || c==n) && a<b << "\n"'
bool is_equal= a==b;
cout<<is_equal<<"\n";
```

EXPRESSIONS AND STATEMENTS

EXPRESSIONS IN C++

- C++ expressions are much like those of other languages.

```
a+3*c
8.0*(double)i+pow(v,3)
sqrt(abs(a-b))
A || B
y > 0.0 && y < 1.0
myfunc(x,y)
```

STATEMENTS

- Indentation is not required but *should be* used!
- Statements are terminated with a semicolon.
- **Code Blocks**
  - Code blocks are multiple statements that are logically a single statement.
  - C++ uses curly braces {} to enclose blocks.
  - Two styles. Pick one and be consistent:

```
if (cond) {
   statements;
}
```

```
if (cond)
{
    statements
}
```

HELLO WORLD

```
#include <iostream>
using namespace std;
int main (int argc, char ** argv) {
  cout<<"Hello world\n";
  return 0;
}
```

EXERCISE

- Write a program that will declare and set variables as indicated and will print the expressions indicated. You may use your "hello world" program as a base.

```
x=17.
Xs=11.
num_1=10
num_2=14

cout<<x<<"\n";
cout<<Xs/x<<"\n";
cout<<(int)Xs/x<<"\n";
cout<<int(Xs)/int(x)<<"\n";
cout<<Xs/x + x<<"\n";
cout<<Xs/(x+x)<<"\n";
cout<<x/num_1<<"\n";
cout<<num_1/num_2<<"\n";
cout<<num_2/num_1<<"\n";
```

EXERCISE

- Declare string variables large enough to hold the indicated strings. Include the header

```
#include <string>
string title="This is a string";
string subtitle="Another string"
cout<<title.size()<<"\n";
string newtitle=title+":"+subtitle;
cout<<newtitle<<"\n";
cout<<newtitle.substr(1,3)<<"\n";
//Quiz:Change "This" to "That" in newtitle
```

C++ LOOPS AND CONDITIONALS

CONDITIONALS

`else if/else if` and `else` are optional

```
  if ( comparison ) {
     code;
    }
    else if ( comparison) {
      more code;
    }
    else  {
     yet more code;
    }
```

SWITCH

- Many `else if`s can become confusing.

```
switch (expression){
          case const value0:
                    code;
                    break;  //optional
          case const value1:
                    code;
                    break;
          case const value2:
                    code;
                    break;
           case const value3:
                    code;
                    break;
            default :   // Optional, usually needs break before
                    code;
        }
```

where “const value” is either something declared `const` or a literal. It must be an integer, or convertible to an integer (so char is acceptable but not string).

SWITCH EXAMPLE

```
switch (chooser) {
  case  (0):
    y=-x2;
    break;
  case (1):
    y=x2+3./x1;
    Break;
  case default
    y=0.;
}
```

FOR LOOP

- `for` executes a fixed number of iterations unless explicitly terminated.

```
for (int i=l;i<=u;i+=s) {
  code;
}
```

  - `i`: Loop variable
  - `l`: Lower bound
  - `u`: Upper bound
  - `s`: Stride. Use `++i` for a stride of 1.

`s` can be negative, in which case `l` must be greater than `u`. For -1 use `--i` or similar.

EARLY EXIT

`break`: leave loop

- `break` is able to break out of *only* the loop level *in which it appears*. It cannot break from an inner loop all the way out of a nested set of loops. This is a case where `goto` may be better than the alternatives.
- `continue`: skip rest of loop and go to next iteration.
- `goto` Syntax (use sparingly):

```
goto Label;
…
Label:
   Code
```

WHILE LOOPS

```
while (<logical expression>) {
      statement
      statement
           …
  }
```

- Remember that your logical expression must become false at some point.

EXAMPLE

```
int x, y, z;
x=-20;
y=-10;
while (x<0 && y<0) {
    x=10-y;
    y+=1;
    z=0;
}
z=1;
```

BREAK/CONTINUE

```
float x=1.;
do while (x>0.0) {
   x=x+1.;
    if (x>=10000.0) break;
    if (x<100.0) continue;
    x+=20.0;
}
```

REPEAT-UNTIL

```
do {
    statement;
    statement;
    ….
    if (<logical expression>) break;
}
```

The `while` always tests at the *top* of the loop.
The `do` ... `if/break` form can test anywhere.

EXAMPLE

```
#include <iostream>
using namespace std;
int main() {
  int x, y, z;
  x=-20;
  y=-10;
  while (x<0 && y<0) {
     x=10-y;
     y=y+1;
     z=0;
  }
  z=1;
  cout<<x<<" "<<y<<" "<<z<<"\n";
  return 0;
}
```

EXERCISE

- Loop from 0 to 20 by increments of 2. Make sure that 20 is included. Print the loop variable at each iteration.
- Start a variable n at 1. As long as n is less than 121, do the following:
  - If n is even, add 3
  - If n is odd, add 5
  - Print n for each iteration. Why do you get the last value?
- Set a float value x=0. Loop from 1 to N inclusive by 1.
  - If the loop variable is less than M, add 11. to x.
  - If x > w and x < z, skip the iteration.
  - If x > 100., exit the loop.
  - Experiment with different values for the variables. Start with N=50, M=25, w=9., z=13.

ARRAYS

TERMINOLOGY

- A *scalar* is a single item (real/float, integer, character/string, complex, etc.)
- An *array* contains data of the **same type** with each scalar element addressed by *indexing* into the array.
- An array has one or more *dimensions*. The *bounds* are the lowest and highest indexes. The *rank* is the number of dimensions.
- C++ does not have arrays as first-class data types. A C-style array is a block of memory. Other options are available in class libraries.

C-STYLE ARRAYS

- Arrays must be declared by type and either by size or by some indication of the number of dimensions.

```
float a[100];
int M[10][10];
```

If a variable is used, it must be a `const`

```
const int N=10;
float z[N];
```

The starting index is always 0, so for a 100-element array the items are number 0 to 99.

ORIENTATION

- Array elements are *adjacent* in memory (this is one of their advantages) and are arranged linearly no matter how many dimensions you declare. If you declare a 3x2 array the order in memory is

  `(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)`

- "Orientation" refers to how the array is stored *in memory*, not to any mathematical properties.
- C++ and most other languages are *row-major* oriented. Some (Fortran, Matlab, R) are *column-major* oriented.
- Loop indices should reflect this whenever possible (when you need loops).
- Move left to right.

  `A(i,j,k)` loop order is `do for i/for j/for k`

INITIALIZING ARRAYS IN C++

- Arrays can be initialized when created
  `float A[3]={10.,20.,30.}`
- Curly braces are required.
- Example

```
#include <iostream>
using namespace std;
int main(int argc, char **argv){
  const int n=5;
  float A[n]={10.,20.,30.,40.,50.};
  for (int i=0; i<n; i++){
    cout<<A[i]<<" ";
  }
  cout << "\n";
return 0;}
```

INITIALIZING (CONTINUED)

- Elements not explicitly initialized will be set to 0.
- Try it:
- In the program on the previous page, try setting

```
float A[n]={};
```

Then try

```
float A[n]={10.,20.,30.};
```

with no other changes to the program

WARNING WARNING WARNING

- C++ happily lets you “walk off” your array.
- Most commonly this occurs when you have variables and you end up attempting to access an element outside of the declared size.
- This usually results in a segmentation violation or sometimes garbage results.
- Example: in your previous code change

```
cout << A[i]<<" ";
```

- To

```
cout << A[i+1]<<" ";
```

MULTIDIMENSIONAL ARRAYS IN C++

- Multidimensional arrays are just "arrays of arrays" in C++.
- They are declared with multiple brackets:
- `float A[2][5];`
- Elements are referenced like

  `A[0][2]`

  `A[i][j]`

- Initialize like

```
A={{1.,2.,3.,4.,5.},
   {6.,7.,8.,9.,10.}};
```

C++ ARRAY PROPERTIES

- The arrays we have discussed are "C style arrays)
- They are just blocks of memory with no added metadata.
- 1D arrays are contiguous in memory but higher-dimensional arrays need not be.
- The name of the array is also a *pointer* to the address in memory of the first (zeroth) element of the array.
- Higher-dimensional arrays cannot be fully dynamically defined.

PASSING ARRAYS TO PROCEDURES

- We can only pass the *pointer* to the first element of the array.
- A pointer is a variable that holds the memory location of another variable.
- Array names are really pointers and in C were usually explicitly so.
- In the procedure's argument list you can declare your array with one empty bracket. For higher-dimensional arrays only the first dimension can be empty; the others must be specified.
- Higher-dimensional arrays generally are declared and passed as pointers, but their dimensions *must* be passed in this case.

EXAMPLE

- In main:

```
float a[100];
```

- In the function

```
float myfunc(float a[], int length)
```

- Invoke the function with

```
myfunc(a, 100);
```

- More about this when we get to functions.

ALLOCATION AND THE NEW OPERATOR

- Arrays may be sized at runtime.
- `int N;`
- `N=30;`
- `float* A=new float[N];`
- The `*` indicates that `A` is a *pointer* to a block of `float` variables. We do not have to use it subsequently, and can still refer to `A` by index, e.g. `A[2]`.

MULTIDIMENSIONAL ARRAYS WITH NEW

- We will only discuss 2d arrays here.
- Two-dimensional arrays are 1-d arrays of pointers to an array.

```
int nrows, ncols;
```

- //Set nrows, ncols by some means

```
float **A;
A=new float*[nrows];
for (int i=0;i<nrows;++i) {
 A[i]=new float[ncols];
}
```

C++ CONTAINERS

- A *container* is a data structure that can contain other types.
- C++ implements most containers as *templates*.
  - This is beyond our scope right now.
- If you need an array with more functionality there is an array container.
  - But it's still fixed size and 1D
- The `vector` container can be sized dynamically.
- Other options include `boost` (most popular), `blitz++` libraries.

BOOST ARRAYS

- Boost is a popular library of extensions and templates for C++
- One of its containers is the multi_array
- To use it, you must install the library (or use a computer where it has been installed for you)
- Templates in general, and boost arrays in particular, can be slow. The example on the next two slides tests this.

BOOST ARRAYS AND C ARRAYS

```
#include <ctime>
#include <boost/multi_array.hpp>

using namespace std;

int main(int argc, char* argv[])
{
    int nrows;
    int ncols;
//  Iterations are to make the time measurable
    const int ITERATIONS = 1000;
    time_t startTime, endTime;

// Set the array dimensions
    nrows=500;
    ncols=500;

// Create the boost array
    typedef boost::multi_array<double, 2> Array2D;
    Array2D boostArray(boost::extents[nrows][ncols]);
// Create the C array
    double **C_Array;
//
```

```
//------------------Measure boost-----------------------------------------
   startTime = time(NULL);
   for (int i = 0; i < ITERATIONS; ++i) {
      for (int x = 0; x < nrows; ++x) {
         for (int y = 0; y < ncols; ++y) {
             boostArray[x][y] = 2.345;
         }
      }
   }

   endTime = time(NULL);
   cout<<"[Boost] Elapsed time: "<<(endTime - startTime)/1000.0<<" sec\n";

//------------------Measure native----------------------------------------
   C_Array = new double* [nrows];
   for (int i=0;i<nrows;++i) {
      C_Array[i]=new double[ncols];
   }
   startTime = time(NULL);
   for (int i = 0; i < ITERATIONS; ++i) {
      for (int x = 0; x < nrows; ++x) {
         for (int y = 0; y < ncols; ++y) {
            C_Array[x][y] = 2.345;
         }
      }
   }
   endTime = time(NULL);
   cout<<"[C style] Elapsed time: "<<(endTime - startTime)/1000.0<<" sec\n";
   return 0;
}
```

EXERCISE

- Write a program to:
- Declare an integer array of 10 elements
- In your program:
  - Print the size of the array
  - Change the fourth element to 11
- Declare a real array of rank 2 (two dimensions) and allocate it with new. Allocate the array and set each element to the sum of its row and column indices.

C++ INPUT/OUTPUT

STREAM IO

- Stream IO allows the compiler to format the data.
- Input
- Header required
- #include <iostream>
  - Read from standard input. Requires whitespace separation.

    ```
    std::cin >> var1 >> var2 >> var3
    ```
  - Write to standard output. `\n` is the end of line marker.

    ```
    std::cout<<var1<<" "<<var2<<" "<<var3<<"\n"
    std::cout<<var1<<" "<<var2<<"
    "<<var3<<std::endl
    ```

READING FROM THE COMMAND LINE

- We can read strings only. You must convert if necessary to a numerical type using *string streams*.

```
#include <iostream>
#include <sstream>
using namespace std;
int main(int argc, char **argv) {
  float value;
  if ( argc>1) {
    stringstream inputValue;
    inputValue<<argv[1];
    inputValue>>value;
  }
  return 0;
}
```

FORMATTED IO

- Formatted input is rarely needed.
- Formatted output permits greater control over the appearance of output. Compilers tend to let stream output sprawl.
- Formatted output also allows programmer control over the number of decimal digits printed for floating-point numbers.

MANIPULATORS

- C++ uses *manipulators* to modify the output of the stream operators `cin` and `cout`.
- A few base manipulators:
- Output
  - `endl` flushes the output and inserts newline
  - `ends` outputs null character (C string terminator)
  - `boolalpha` true/false printed for Booleans
  - `left/right/internal` left/right/internal for fillers.
- Input
  - `ws` reads and ignores whitespace
  - `skipws/noskipws` ignore/read initial whitespace as characters

HEADER IOMANIP

- `#include <iomanip>`
- These manipulators stay in effect in a given output stream until cancelled.
- `setw(n)` Set width output quantity will occupy
- `setprecision(n)` Set number of places printed for floating-point numbers
- `fixed/scientific` Fixed-point format or scientific notation format
- `setfill(c)` Set a filler character `c`
- `setbase(n)` Output in base `n` (options are 8, 10, or 16, or 0 which reverts to default of decimal).

EXAMPLE

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
  float x=.00001, y=17., z=10000.;
  cout<<setprecision(16)<<z/y<<"\n";
  cout<<setw(20)<<setfill('*')<<left<<z<<"\n";
  cout<<scientific<<x<<" "<<z<<"\n";
  cout<<scientific<<x<<" "<<fixed<<z<<"\n";
}
```

EXERCISE

- Write a program that computes pi using a trig identity such as π=4*atan(1). Remember
- #include <cmath>
- To switch between float and double easily, use
  - typedef float real;
- Switch "float" to "double" to change.
- Declare variables as
  - real x;
- Using single precision, print pi in
  - Scientific notation
  - Scientific notation with 8 decimal places
- Repeat for double precision, print scientific notation to 12 places

EXERCISE

- In an “infinite” while loop:
- Request an integer from the user without advancing to a new line, e.g.
- “Please enter an integer:” <then read integer>
- If the integer is 1, print “zebra”. If it is 2, print “kangaroo”. If it is anything else except for zero, print “not found”. If it is 0, exit the loop.

C++ FILE IO

FILE STREAMS

- Standard streams are automatically opened. Other files must be opened explicitly.
- Files can be input streams (ifstream), output streams (ofstream), or either/both (fstream).
- `#include <fstream>`

or

- `#include <ofstream>`
- `#include <ifstream>`

as needed.

OPEN

- First a stream object must be declared.

```
ifstream input;
```

- Then the stream can be attached to a named file

```
input.open(inFileName);
```

  - This assumes the file exists and is opened for reading only.

- For output use

```
ofstream output;

output.open(outFileName);
```

- This file will be emptied if it exists or created if it does not exist, and will be opened in write-only mode.

MODIFIERS

- We can control the characteristics of the file with modifiers

`ios::in` Open for input (read). Default for ifstream.

`ios::out` Open for output (write). Default for ofstream.

`ios::binary` Open as binary (not text)

`ios::app` Append

`ios::trunc` If file exists, overwrite (default for ofstream)

Use a pipe `|` to combine them

```
ofstream.open("myfile.dat",ios::binary
| ios::app);
```

INQUIRING

- All inquiry methods return a `bool` (Boolean).
- To check whether a file is open

```
infile.is_open()
```

- To check whether a file opened for reading is at the end

```
infile.eof()
```

- Generic testing

```
mystream.good()
```

CLOSE

- Much of the time, it is not necessary to close a file explicitly. Files are automatically closed when execution terminates.
- If many files are opened, it is good practice to close them before the end of the run.
- `mystream.close();`

REWIND

- An open unit can be rewound. This places the *file pointer* back to the beginning of the file.
- The default is to rewind a file automatically when it is closed.
- These are C-style functions and are in `<cstdio>`

```
rewind(mystream)
```

- You can also seek to position 0

```
fseek(mystream,0,SEEK_SET)
```

- `rewind` clears the end-of-file and error indicators, whereas `fseek` does not.

READING FROM A FILE

- Frequently we use `getline`
- Example

```
string line;
//note implicit open
ifstream mystream("datafile.txt");
if ( mystream.is_open()) {
  while (getline(mystream,line)) {
    //do something with line
  }
}
else {
  cout <<"Unable to open file";
```

GETLINE

- Getline's name is a little misleading.
- Getline actually reads to a delimiter. The default delimiter is newline `\n`.
- getline(istream,string,char delim)
- getline(istream,string)
- The delimiter character is discarded from the string.
- Example:

```
cout<<"Enter your name:";
getline(cin,name);
```

READING A CSV FILE

- We often need to read files where each line contains several fields separated by a comma or other delimiter. Example: read four values from each line for 200 lines, ignoring the second column values.

```
const int nobs=200;
float bf[nobs], wt[nobs], ht[nobs];
string line;
ifstream fin("datafile.txt");
if (fin.is_open()) {
     while (getline(fin,line)) {
          stringstream lineStream(line);
          string * linevals=new string[4];
          int index=0;
          while ( getline(lineStream,linevals[index],',') ) {
               ++index;
          }
          stringstream ssbf, sswt, ssht;
          ssbf<<linevals[0];
          ssbf>>bf[lineCount];
          sswt<<linevals[2];
          sswt>>wt[lineCount];
          ssht<<linevals[3];
          ssht>>ht[lineCount];
          lineCount++;
          }
}
else {
     cout <<"Unable to open file";
     return 1;
}
```

WRITING TO A FILE

- Write to a file much like to a standard stream.

```
ofstream out("outfile.txt");
out<<"column1,column2,column3\n";
for (int i=0;i<nlines;++i) {
  out<<var1[i]<<","<<var2[i]<<","<<var3[i]<<"\n';
}
```

EXERCISE

- Write a program that creates a file mydata.txt containing four rows consisting of
- 1, 2, 3
- 4, 5, 6
- 7, 8, 9
- 10, 11, 12
- Rewind the file and read the data back. Write a loop to add 1 to each value and print each row to the console.

SUBPROGRAMS

WHAT IS A SUBPROGRAM

- A subprogram is a self-contained (but not standalone) program unit. It performs a specific task, usually by accepting *parameters* and returning a result to the unit that invokes (calls) it.
- Subprograms are essential to good code practice. Among other benefits, they are
  - Reusable. They can be called anywhere the task is to be performed.
  - Easier to test and debug than a large, catch-all unit.
  - Effective at reducing errors such as cut and paste mistakes.

FUNCTIONS AND SUBROUTINES

- Functions take any number (up to compiler limits) of arguments and return one item. This item can be a compound type.
- Functions must be declared to a type like variables.
- Subroutines take any number of arguments (up to the compiler limit) and return any number of arguments. All communication is through the argument list.
- Strictly speaking, all subprograms in C++ are functions, but the ability to declare a `void` return "type" means some are effectively subroutines. Subroutines communicate only through their parameter list.
- In C/C++ either the function or its *prototype* must appear before any invocation.

FUNCTIONS

The return value is indicated by the `return` statement.

```
<type> myfunc(<type> param1,<type>
     param2,<type> param3,<type> param4){
   statements
   return aResult;
}
```

INVOKING FUNCTIONS

- Function
  - Invoke by its name
  - `x=myfunc(z,w)`
  - `y=c*afunc(z,w)`

  A function is just like a variable except it cannot be an *lvalue* (appear on the left-hand side of =)

PASSING BY VALUE OR BY REFERENCE

- Most parameters in C++ functions are passed by *value*. The compiler makes a copy and places it into the corresponding function variable.
- C++ can pass by *reference* (more easily than C). This means that the subprogram receives a pointer to the location in memory of the variable.
- When passing by reference, if that argument is changed by the subprogram it will be changed in the caller as well. This is a **side effect**.
- Subroutines operate *entirely* by side effects.
  - Sometimes this is not called a “side effect” when it is intentional, only when it is unintentional.

"SUBROUTINES" IN C++

- The & indicates we are passing by reference. This is the value that will be modified in this example.

```
void mysub(<type> param1,<type>
     param2,<type> &param3){

         statements

}
```

- Invoke with its name

```
mysub(param1, param2, param3);
```

EXERCISE

1. Write a function that computes Euclidean distance between points `x1,y1` and `x2,y2`. Include the `<cmath>` header to get the `sqrt` intrinsic that you should use.

```
#include <cmath>
```

Write the main program to call this function for

```
x1=-1, y1=2, x2=3, y2=5
x1=11,y1=4, x2=7, y2=9
```

2. Given two points `x1,y1` and `x2,y2`, write a subroutine to determine which is closer to a third point `x3,y3`. It should print a message. You can pass in the points and call the Euclidean distance function from the subroutine, or you can pass in the two distances. (The former would be better programming but if you feel uncertain please go ahead and compute distances separately for now.) Test with

```
x3=10, y3=5
```

PASSING ARRAYS TO SUBPROGRAMS

- One-dimensional arrays may be passed as pointers or with empty square brackets []. The size must be passed as well.

```
float mean(float A[], int n);
myMean=mean(A,n);
```

- This is equivalent to

```
float mean(float *A, int n);
```

- C-style arrays are always passed by reference.
- Containers such as vectors may be passed either by copying or by reference.

EXAMPLES

```
#include <iostream>
using namespace std;

float mean(float A[],int n){
    float sum=0;
    for (int i=0;i<n;++i){
        sum+=A[i];
    }
    return sum/(float)n;
}

float mean2d(float **A, int n, int m){
    float sum=0;
    for (int i=0;i<n;++i) {
        for (int j=0;j<m;++j) {
            sum+=A[i][j];
        }
    }
    return sum/(float)(n*m);
}
```

EXAMPLES (CONTINUED)

```
int main(int argc, char **argv) {


    int n=6, m=4;
    float *A=new float[n];
    float **B=new float*[n];
    for (int i=0;i<n;++i) {
        B[i]=new float[m];
    }


    for (int i=0;i<n;++i) {
        A[i]=i+1;
        for (int j=0;j<m;++j) {
            B[i][j]=i+j+2;
        }
    }
    float mymean=mean(A,n);
    cout<<mymean<<"\n";


    float mymean2d=mean2d(B,n,m);
    cout<<mymean2d<<"\n";


return 0;
}
```

LOCAL ARRAYS

- Arrays that are local to a subprogram may be sized using an integer passed to the subprogram.
- Local array memory **must** be released in the subprogram or a *memory leak* will result.
- Wrong:

```
float newmean(float A[],int n){
    float *B=new float[n];
    float sum=0;
    for (int i=0;i<n;++i){
        B[i]=A[i]+2;
        sum+=B[i];
    }
    return sum/(float)n;
}
```

DELETE OPERATOR

- The `delete` operator releases memory allocated by `new`.
- In principle, each `new` should have a corresponding `delete`. In main programs, however, the memory will be automatically released when the program ends. Best practice is to always use `delete`, however.
- It is essential to pair `delete` with `new` in subprograms.

```
float newmean(float A[],int n){
    float *B=new float[n];
    float sum=0;
    for (int i=0;i<n;++i){
        B[i]=A[i]+2;
        sum+=B[i];
    }
     delete [] B;  //frees all memory associated w/ B
    return sum/(float)n;
}
```

PROTOTYPES

- Traditionally, the entire function bodies were placed at the top of the file where they were used.
- Modern practice in C++ is to write prototypes separate from bodies.
- Prototypes enable the compiler to check that the *number* and *type* of the argument list in invocations agrees with the declared parameter list.
- Prototypes are frequently collected into files ending in `.h` (header files) with function bodies in a corresponding `.cxx` (or `.cpp`) file. The prototypes are the *interface* and the bodies are the *implementation*.
- Interface+implementation makes it easy to reuse the code.

EXAMPLE

- Note: in prototypes we need only specify number and type of the parameters.
- means.h

```
float mean(float *,int);
float mean2d(float **, int, int);
```

- means.cxx

```
float mean(float A[],int n){
    float sum=0;
    for (int i=0;i<n;++i){
        sum+=A[i];
    }
    return sum/(float)n;
}

float mean2d(float **A, int n, int m){
    float sum=0;
    for (int i=0;i<n;++i) {
        for (int j=0;j<m;++j) {
            sum+=A[i][j];
        }
    }
    return sum/(float)(n*m);
}
```

USING PROTOTYPES

- Headers not in the system are usually specified in quotes rather than angle brackets.

```
#include <iostream>
#include "means.h"
using namespace std;

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

    int n=6, m=4;
    float *A=new float[n];
    float **B=new float*[n];
    for (int i=0;i<n;++i) {
        B[i]=new float[m];
    }

    for (int i=0;i<n;++i) {
        A[i]=i+1;
        for (int j=0;j<m;++j) {
            B[i][j]=i+j+2;
        }
    }
    float mymean=mean(A,n);
    cout<<mymean<<"\n";

    float mymean2d=mean2d(B,n,m);
    cout<<mymean2d<<"\n";
```

COMPILING AND LINKING

- We now need more than one file to build the executable.
- Under Linux we can compile with

```
g++ -c means.cxx
g++ -c main.cxx
g++ -o means main.o means.o
```

EXERCISE

- Correct your previous exercise with Euclidean distance function and subroutine to use prototypes. You may use a single file for this exercise (prototypes at the top following `include` and `using` lines, bodies after main).

DEFAULT ARGUMENTS

DEFAULT ARGUMENTS

- Subprogram may take default (optional) arguments. Such arguments need not be passed. If they are passed, they take on the passed value. They are declared by specifying a default value.

```
int myfunc(float x, float y,float
     z=0.,float w=1.);
```

or in a prototype

```
int myfunc(float, float,
     float=0.,float=1.);
```

- Arguments that are not optional are *positional* and their order matters. Positional arguments must precede all optional arguments.

USING DEFAULT ARGUMENTS

- The call to the previously-defined subroutine could be

  ```
  n=myfunc(a,b)
  ```

  in which case c and d would have their default values. The call could also be

  ```
  m=myfunc(a,b,c)
  ```

  or

  ```
  l=myfunc(a,b,c,d)
  ```

  depending on how many of the default arguments needed to be passed.

- Note: C++ does not support named (keyword) arguments, unlike Python, Fortran, and some other languages.

SCOPE

VARIABLE SCOPE

- In C++, scope is defined by the *code block*. Code blocks are enclosed in curly braces {}
- A scope unit may have a variable named `x`, and a function may also have a variable named `x`, and if `x` is not an argument to the function then it will be distinct from the `x` in the calling unit.

```
float x=20.
float z=sub(x)
   etc.

float sub(float y){
   float x=10.
   float y=30.
}
```

CODE BLOCK EXAMPLES

- Loops

```
for (int i=0;i<4;++i) {
    cout << i << "\n";
}
cout << i << "\n";
```

- Results in: `warning: name lookup of ‘i’ changed for ISO ‘for’ scoping [-fpermissive]`
- Free-standing blocks

```
j=12;
{j=13;
  cout<<j<<"\n";
}
cout<<j<<"\n";
```

COMPILERS, LINKERS, AND MAKE

BUILDING AN EXECUTABLE

- The compiler first produces an *object file* for each *source file*. In Unix these end in `.o`
- Object files are binary (machine language) but cannot be executed. They must be linked into an executable.
- If not told otherwise a compiler will attempt to compile and link the source file(s) it is instructed to compile.
- For Unix compilers the `-c` option suppresses linking. The compiler must then be run again to build the executable from the object files.
- The option `-o` is used to name the binary something other than a.out

LINKERS AND LIBRARIES

- When the executable is created any external libraries must also be linked.
- The compiler will search a standard path for libraries. On Unix this is typically `/usr/lib, /usr/lib64, /usr/local/lib, /lib`
- If you have others you must give the compiler the path. `-L` followed by a path works, then the libraries must be named `libfoo.a` or `libfoo.so` and it is referenced `-lfoo`
- Example:

```
gfortran -o mycode -L/usr/lib64/foolib mymain.o mysub.o -lfoo
```

MAKE

- `make` is a tool to manage builds, especially with multiple files.
- It has a rigid and peculiar syntax.
- It will look for a `makefile` first, followed by `Makefile` (on case-sensitive systems).
- The `makefile` defines one or more *targets*. The target is the product of one or more *rules*.
- The target is defined with a colon following its name. If there are *dependencies* those follow the colon.
- Dependencies are other files that are required to create the current target.

TARGETS AND RULES

- Example:

```
myexec: main.o module.o
<tab>gfortran -o myexec main.o module.o
```

- The tab is *required* in the rule. Don't ask why.
- Macros (automatic targets) for rules:
- `$@` the file name of the current target
- `$<` the name of the first prerequisite

VARIABLES AND COMMENTS

- We can define variables in makefiles
- `F90=gfortran`
- `CXX=g++`
- We then refer to them as `$(F90)`, `$(CXX)`, etc.
- Common variables: `F90`, `CC`, `CXX`, `FFLAGS`, `F90FLAGS`, `CFLAGS`, `CXXFLAGS`, `CPPFLAGS` (for the preprocessor), `LDFLAGS`

SUFFIX RULES

- If all .cxx (or .cc or whatever) files are to be compiled the same way, we can write a *suffix rule* to handle them.
- It uses a *phony target* called .SUFFIXES.

```
.SUFFIXES: .cxx .o
    $(CXX) -c $(CXXFLAGS) -c $<
```

PATTERN RULES

- An extension by Gnu make (`gmake`), but nearly every `make` is `gmake` now.
- Similar to suffix rules.
- Useful for Fortran 90+:

```
%.mod: %.o
```

- Pattern for creating the .o:

```
%.o: %.f90
	$(F90) $(F90FLAGS) -c $<
```

EXAMPLE

```
PROG =  bmi

SRCS =  bmi.cxx bmistats.cxx stats.cxx

OBJS =  bmi.o bmistats.o stats.o

LIBS =

CC = gcc
CXX = g++
CFLAGS = -O
CXXFLAGS = -O -std=c++11
LDFLAGS =
all: $(PROG)

$(PROG): $(OBJS)
        $(CXX) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

.PHONY: clean
clean:
        rm -f $(PROG) $(OBJS) *.mod

.SUFFIXES: $(SUFFIXES) .c .cpp .cxx

.c.o:
```

ABSTRACT TYPES

DERIVED TYPES

- In C++ abstract types are called *structs*.
- The syntax is extremely simple (ptype stands for a primitive type)

```
struct mytype {
  <ptype> var1;
  <ptype> var2;
};
```

- Example

```
struct Employee {
  string name, department;
  int ID;
  float salary;
};
```

- Each variable belonging to a struct is called a *member*.
- The variables declared as a struct are often called *instances* of that struct.
- Note: it is customary for the name of a struct (or class) to be capitalized, or to use "camel case."

DECLARING TYPES AND ACCESSING FIELDS

```
Employee fred, bill, susan;
```

To access the fields of the type use the name of the type, a decimal point as a separator, and the name of the field.

```
fred.name="Frederick Jones";
fred.ID=1234;
fred.department="Accounting";
fred.salary=75200.00;
```

STRUCTS IN STRUCTS

- Struct members may be instances of other structs.

```
struct Address {
   string streetAddress;
   string city, state;
   int zipCode;
};
struct Employee {
  string name, department;
  int ID;
  float salary;
  Address homeAddress;
};
```

THE ARROW OPERATOR

- As for other types, variables can be declared pointer to struct

```
Employee *jane;
```

- This is particularly common when passing struct (and class) instances to functions, to avoid a copy.
- When using a pointer, the `.` field separator is replaced with the *arrow operator*

```
jane->name="Jane Smith"
```

VECTORS

- *Containers* are data structures that can be filled with any type (or at least multiple ones). Several are available but here we will only discuss the **vector**.
- Using a vector requires including its header

```
#include <vector>
```

- A vector is a *template* so it must be told what type it is going to be using.

```
std::vector<float> V;
```

`std::` can be omitted if `using namespace std` (but avoid the `using` statement in `.h` files).

We'll assuming the `std` namespace for the notes.

INITIALIZING VECTORS

- Vectors are similar to one-dimensional arrays
  - They represent an ordered sequence of elements
  - Elements are accessed by integers 0…N-1 (for size N)
- But unlike arrays, vectors are dynamic.
  - It's possible to enlarge and shrink them.
- Initializing vectors
  - Loops (like an array)

```
vector<float> V(N);
for (int i=0;i<N;++i) {
   V[i]=(float) i;
}
```

  - Dynamic Sizing

```
vector <float> V={};
for (int i=0;i<N;++i) {
   V.push_back(i);   //appends i at the end of V
}
```

  - Initializer List (C++11)

```
vector<float> V={1., 2., 3., 4., 5., 6.}
```

USEFUL VECTOR METHODS

- For a vector V:
  - `V.push_back(item)`
    - Append item to V
  - `V.at(index)`
    - Access [index] with bounds checking ([index] does no checking)
  - `V.start()`
    - Starting point for iterator
  - `V.end()`
    - End point (beyond last element) of iterator
  - `V.size()`
    - Number of elements
  - `V.clear()`
    - Empty V and make it size 0

VECTORS AND STRUCTS

- Vectors can be members of structs

```
struct Data {
  int nobs;
  vector<float> obs;
}
```

- Vector elements can be struct instances

```
vector<Data> dataList;
```

EXAMPLE

- This struct encapsulates a set of observations for birds denoted by their common name.

```
struct birdData {
//Input values.
    string commonName;
    vector<float> observations;
};
```

CLASSES

OBJECT-ORIENTED PROGRAMMING

- An **object** is a data structure which has associated *data* (variables) and *behaviors* (subprograms).
- Objects work on their own data, communicating with outside units through an interface.
- Objects *encapsulate* related concepts and keep them unified.
- In most object-oriented programming languages, objects are represented by *classes*.

OOP TERMINOLOGY

- An *instance* of a type or class is a variable of that type/class.

```
Myclass A, B;
```

`A` and `B` are instances of Myclass

- A variable that is a member of the type/class is often called an *attribute*.
- A *method* is a subprogram that is a member of a class.
- An invocation of a method is often called a *message*.

DATA HIDING

- One of the main purposes of OOP is to prevent outside units from accessing members in an uncontrolled way.
- Making a member *public* "exposes" it and allows anything that uses the class direct access to the member.
- Typically, attributes are *private* and methods are public.
- Methods can be written to obtain or change the value of an attribute.

ACCESS CATEGORIES

- public
  - Accessible directly through an object
  - myobj.var1
- private
  - Accessible only within the class or to "friend" classes. Must be communicated outside the class through an accessor ("getter") and changed through a mutator ("setter"). Default for a C++ class member.
- protected
  - private within the class, accessible to "friend" classes and to descendant classes.

CLASSES IN C++

- We define a class with the keyword class

```
class Myclass {
  public:
    double var1;
    double var2;
    int  ival;
};
```

Note: no methods in this class and all attributes are public. So it's equivalent to a `struct`.

METHODS

- The class definition contains only the prototypes of the methods.

```
class MyClass {
  public:
    double var1, var2;
    int var3;

    double function1(double x, double y);
    double function2(double x);
};
```

METHODS

- The methods are defined with class::function

```
double MyClass::function1(double x,
                          double y) {
  return x+y;
}
double MyClass::function2(double x) {
 var1=x*var2;
  return;
}
```

CONSTRUCTORS AND DESTRUCTORS

- A class always has a *constructor* and a *destructor*. If you don't write them the compiler will try to do it for you.
- The constructor is automatically called when an object (instance) of the class is created.
  - If you declare a variable of the type this calls the constructor
  - If you declare a variable of type pointer-to-class the constructor is called by the new operator.
- The destructor is called when the object is released.
- The constructor has the same name as the class.
- The destructor has the same name as the class but preceded by ~
- Constructors usually must be public.

CONSTRUCTOR AND DESTRUCTOR

```
class MyClass {
  public:
    double var1, var2;
    int var3;
    MyClass(double v1, double v2, int
    v3);
    ~MyClass(); // destructors never
    have arguments
    double function1(double x, double
    y);
    double function2(double x);
};
```

METHODS

```
MyClass::MyClass(double v1, double v2, int v3) {
      var1=v1; var2=v2; var3=v3;

MyClass::~MyClass(){
//Not much to do in this example
}
double MyClass::function1(double x, double y) {
  return x+y;
}
double MyClass::function2(double x) {
  var1=x*var2;
  return;
}
```

EXAMPLE

```
/*
* testclass.cxx *
*/
class MyClass {
    public:
        double var1, var2;
       int var3;
       MyClass();
       ~MyClass();

    private:
        double privatevar;
};
MyClass::MyClass(double var1, double var2, int var3){code}
MyClass::~MyClass(){}
#include <iostream>
int main(int argc, char **argv){
    MyClass mytest;
    mytest.var1=11.; mytest.var2=25.; mytest.var3=5;
    mytest.privatevar=13.;    ← ILLEGAL
return 0;}
```

CORRECT (AND SQUISHED)

```
/*
* testclass.cxx
*
*/
class MyClass {
     public:
        double var1, var2;
        int var3;
        MyClass();
       ~MyClass();
        void set_privatevar(double value);
        double get_privatevar();
    private:
        double privatevar;        };


MyClass::MyClass(double v1, double v2, int v3){var1=v1;var2=v2;var3=v3;}
MyClass::~MyClass(){}
void MyClass::set_privatevar(double value) {privatevar=value;}
double MyClass::get_privatevar(){return privatevar;}
#include <iostream>
using namespace std;
int main(int argc, char **argv){
      MyClass mytest;
      mytest.var1=11.; mytest.var2=25.; mytest.var3=5;
      mytest.set_privatevar(13.);
     cout<<mytest.get_privatevar()<<"\n";
return 0;}
```

THIS

- C++ does not pass the instance variable explicitly (but it is there).
- If you need access to it in a method, use the `this` parameter.
- `this` is a pointer so requires the arrow operator.
- One example is using the same variable name as an argument and an attribute.

```
Myclass::Myclass(x,y,z) {
        this->x=x;
        this->y=y;
        this->z=z;
}
```

MEMBERS IN METHODS

- If you change a class member variable (attribute) in a method do not return that variable.
- Methods that do nothing but set one or more attributes (mutators) do not return anything.

```
void Myclass::setx(x) {
      this->x=x;
      return;
}
```

- Anything delivered outside the instance is returned. E.g. accessors

```
float Myclass::getx(){
     return x;
}
```

EXERCISE

- Write a class Atom that contains the following attributes:
  - Element symbol
  - Element name
  - Atomic mass
  - Atomic number
- The methods should be
  - Constructor to set attributes
  - Compute and return the number of neutrons from the mass and number (n=mass-number)

INHERITANCE AND POLYMORPHISM

- Classes can inherit from parent classes.
- New classes are called *derived classes* or *child* classes.
- Multiple parents are allowed, but this is generally discouraged.
- Attributes are not inherited if they are declared `private`. They must be public to be transmitted.
- Constructors are a little more complicated with inheritance.
- We will show just one simple example to illustrate.

ATTRIBUTE INHERITANCE

- The child type inherits all the attributes of its parent.

```
Child billy;
```

- The child inherits the constructor and accessor from the parent.
- But `age` does not refer back to the parent, since that variable occurs only in the child.

EXAMPLE

```
#include <iostream>
using namespace std;

class Parent {
  protected:
    int myID;
    string name;
  public:
    Parent(string name, int myID);
    string getName();
    int getID();
};

Parent::Parent(string name, int myID) {
   this->name=name;
   this->myID=myID;
}

string Parent::getName() { return this->name; }
int Parent::getID() { return this->myID; }
```

EXAMPLE (CONTINUED)

```
class Child: public Parent {
   private:
     int age;
   public:
     Child(string name, int myID, int age);
     int getAge();
};

Child::Child(string name, int myID, int age) : Parent(name, myID) {
   this->age=age;
}

int Child::getAge() { return this->age; }

int main() {
  Child billy("Bill",345,20);
  cout<<billy.getName()<<" "<<billy.getID()<<" "<<billy.getAge()<<"\n";
  return 0;
}
```