CS 294-73
Software Engineering for Scientific Computing

Lecture 2: Our first C++ programs: Hello World, and others; pointers and memory management

Notation

- A line that starts with a “>” character means this is something you type at a shell prompt
  - > ls
  - > cd
- Text that appears in a file or in your editor or debugger will be shown in fixed courier font and not bulleted

```
#! /bin/bash
cd /dada &> /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

Some comments

- The web is your friend.
  - Tools, documentation, alternative points of view.
  - Example: google “C++ reference card”
- Wikipedia is your friend.
  - Will annotate in green topics where Wikipedia provides useful overviews at an appropriate technical level, or history. For example, to get a flavor of the history of what we will discuss in the next few lectures, look up C , C++ .

What makes up a C++ program ?

- Variables ("types"), and associated literals (1,1.5e-6, "dog").
- Functions (user-defined functions, external libraries)
  - Arguments / return values (`y = f(x)`).
  - Control structures (e.g. loops, conditionals)
  - Scope (who sees what)
- "Go button" (aka main)

We will go through this in two passes:

- The "C subset" of C++.
- The language features that make C++ different (and far more powerful) than C. Types, functions, and scope become more powerful and flexible.

Built-in types (AKA primitive types or primitive object data (POD))

- Defined by the language.
- Have internal representation that is not alterable by a program
- C has just 4 built-in types
  - `int, char, float, double`
- C++ adds `bool` (values can be `true` or `false`)
- Can also include modifiers on int types
  - short, long, long long, signed, unsigned
- All built-in types are represented in memory as a contiguous set of *bytes* (byte = 8 bits, bit=[0,1])
  - the C language command `sizeof` will tell you how many bytes
  - `sizeof(char)` is defined to be 1 for all implementations of C/C++

Arrays

- Any of the built-in types in C++ can be declared as an array of that type

```
float e[16];
```

- A float is 4 bytes long, so

```
sizeof(e[16]) == 64;
```

- the [ ] tokens are used both in the declaration of an array, as well as in the accessing of an array element

```
e[i] = i*2;
```

- notice that C/C++ is a *free-format* language
  - the compiler doesn't care about whitespace or carriage returns or tabs
  - spacing and alignment and formatting conventions are up to the developer and the style they find most agreeable.

char

- A one byte integer ? your window to ASCII ?
  - both actually
    -128 to 127
- The ASCII tables
  - 64 @
  - 65 A
  - 66 B
  - .
  - 96 `
  - 97 a
  - 98 b

- As you can guess, working with characters one at a time would be painful

Strings

```
char  myString[] = “some text”;
```

- Not new built-in type, but there is something special
  - We get some snazzy initialization syntax for array of char
  - `sizeof(myString) == 10`
  - An array of char is called a string because there is a an extra zero at the end. This is referred to as a null-terminated array.
  - Many functions and operations in C use strings.

Hello World

- Create a file named helloWorld.cpp with this in it (we call this a *source file*)

```
#include <cstdio>
int main(int argc, char* argv[])
{
  int errCode;
  /* This next line of code will print something*/
  errCode = printf("Hello World\n");
  return errCode;
}
```

- At your shell prompt execute the following command (we call this *compiling*)

```
> g++ helloWorld.cpp
```

- Verify this command created a file call "a.out"
- Run this program

```
> ./a.out
Hello World
```

Compiled vs. Interpreted

- C/C++ is a *compiled* programming language.
  - In the previous example you saw that you use another program (a compiler) to turn your file into another *special* kind of file.
  - The second file (a.out) is them run afterwards to see what your program does.
  - The hallmarks of a compiled language are
    - This two step process.
    - The second file is in machine language (also called “a binary”, or “an executable”).
- This is contrasted with *interpreted* languages.
  - python, matlab scripts, perl, are examples of interpreted languages.
  - All the user works with are source files.
  - Interpreted languages need another program to read and execute them (matlab scripts are run inside matlab, python is run inside the python virtual machine, shell scripts are executed by the shell program)
- We’ll come back to this pros/cons at a later date.

Back to My First C program ♭

- Let's dissect what we wrote.
- #include <cstdio>
  - "#" starts a directive to the *preprocessor* stage of compilation
  - In this case the directive is an "include" command
    - Take the contents of the file following this word and include it here in my program as if I has inserted it in my editor.
  - The cstdio file provides a definition of the printf function
    - Printf is a function for I/O that is part of the *standard library* and specified by the C/C++ language standard. Printf only prints strings, but provides the ability to turn POD into strings.
    - The standard library comes with the compiler.
  - You can use include to make your programming more efficient
    - This can also lead to compiling errors that are mysterious, so, you are only going to use include how we tell you.

♭ : We are using C++ syntax, but no C++ features yet

main

- `int main(int argc, char* argv[])`
- Every C/C++ program has a main function.
  - main is a function: declaration includes arguments, return values.
  - When a program is run, the calling program scans your executable file and when it finds the `main` function it executes it.
    - main defines the entry point of any executable.
  - The program that runs your program is what passes in the arguments, and receives the return value (in this case, an integer).
    - your computer has programs calling programs…and that's it.
      - functions calling functions
  - The first argument is the name of your executable. By typing

```
> ./a.out
```

    - argc the number of arguments (`argc` = 1)
    - argv a array of size 'argc' describing with characters in each argument (`argv[0]` = "`./a.out`")

main

- `int main(int argc, char* argv[])`
- It is easiest to understand the three parts of the main function with another example program

```
#include <cstdio>
int main(int argc, char* argv[])
{
  for(int i=0; i<argc; i++)
    { printf("%s ", argv[i]); }
  printf("\n");
  return argc;
}
```

- Now save this file, compile it, then execute it like so

```
>./a.out sandy wendy –i inescapable=tragedy
./a.out sandy wendy –i inescapable=tragedy
>echo $?
5
(look up “echo”, “regular expression” ($,?))
```

Scoping { }

- C/C++ indicates a change in scope with the curly braces
  - { increase scope depth by 1
  - } decrease scope depth by 1
  - a correct compiled program leaves main with scope==0
  - local variables disappear when they go “out of scope”
  - different computer languages can have different scoping rules.
- Scoping is also used for delimiting the BEGIN and END of language features
  - where a function starts and ends
  - the extent of a loop or conditional statement
  - our two example programs have used scoping twice
- Variables that live in scope==0 are referred to as global and can be accessed at any point in a program.

Variables (Type systems)

```
int errCode;
```

- This is a declaration that
  - We have a variable named “errCode”
  - That variable is an int (integer).
- C++ is a strongly-typed, statically typed language – all variables must be declared before they are used, so the types are known at compile time. Not all languages are statically-typed (e.g. Python).
  - Compiler catches errors.
- In C++, declarations can occur anywhere prior to their use, as long as they are within scope. This includes simultaneous declaration and assignment:

```
int errCode = printf(”Hello World\n");
```

Calling a function

```
printf("Hello World \n");
```

- The compiler parses this line of your program, sees

  ```
  text(          );
  ```

  - That must be a function call.
  - Put a *jump* instruction into `a.out` with destination named `printf.` Put the literal string `"Hello World \n"` into `a.out` in the right location for `printf` to see it as its input argument.
  - The *jump* instruction tells your processor to start executing the code at a new location.
- Why does the compiler think such a destination exists ?
  - <cstdio> told the compiler about several functions, one of them is printf
  - <cstdio> C Standard Input Output is part of the C Standard Library
  - The C/C++ compiler knows where this library is on your system.

Return

```
return errCode;
```

- Our `main` function ends with a call to the C command `return`
- The `return` is matched up with the leading `int` in main's declaration

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

- When we told the compiler about main, it's signature said "I will be returning an integer". The `return` command is how you do this.
  - When return executes, the optional return value is placed in the special place this operating system expects it, and hands control back to the calling function.
- There is an implied empty `return` command at the closing scope } of all functions.
  - A jump instruction was used to call the function. The return command tells the processor to go back can continue with the instruction following the jump in the original sequence.

The compiler has reached the end of helloWorld.cpp

- Several steps in the compilation have not been shown in this simple example.
- As you can imagine, once you have a big program you don't want one giant source file.
- One option for a project as it gets bigger: Start using the `#include` directive.
  - Most interpreted languages rely on forms of #include
  - Shell scripts, python, matlab, etc.
- In a compiled language we can place different parts of our program in different files, compile them separately into their own *objects*, and then stitch them together with a tool called the *linker*.
  - This is appealing enough that some languages combine interpretive and compiled language features (Java, Python, D)
  - For now we will continue with single-file programs

Comments

```
/* This next line of code will print something*/
```

- Called a “C-style” comment
  - Tells the compiler to ignore everything in between the /* and the */ tokens
- `//  This next line of code will print something`
- C++ style comment.
  - Tells compiler to ignore everything from the // token to the next carriage return
- Comments are used to communicate with our human developers.
  - Notes to yourself to remind you what you were thinking when you wrote this code → anything not obvious from a quick perusal of the code.
  - Notes to other developers that will end up in your source code at some future time and must understand it → especially descriptions of the arguments to the functions you write!

Next C Program

```
#include <cstdio>
int main(int argc, char* argv[])
{
  float f=80000023, e[16];
  double d;
  d=80000023;
  int loopI=0;
  for(int j=0; loopI<3; loopI++, j=j-5)
    {
      int result = loopI * j;
      f+=loopI;  d+=loopI;  e[5]+=j;
      printf("%d ",result);
    }
  printf("\n %d %10.1f %10.1f %10.5f \n",loopI, f, d, e[5]);
}
```

- Built-in types: float, double, int, char, arrays [ ]
- What is initialized? When?
- `for` loop syntax
- What is the program output?
- More printf power
- Just what happens with a float or a double ?
- If we try to print result outside the loop body, we get a compiler error.

First thing you should notice!

- Where are the comments?
- What is this program even supposed to be doing ?
- Variables with names like “e” and “f” ?
  - What are those names telling you when you come back to this program a week from now ?
- Mixed initialization styles
- Packing commands on single lines

- Forgiveness is begged in some of these to pack things in a single slide with large fonts.
- You will not have this constraint in your homeworks, and in your programming future.

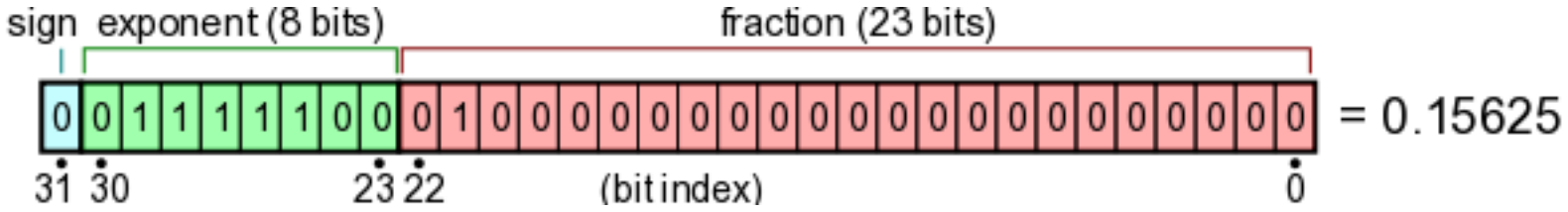
Signed Integer. what is it ?

- `sizeof(signed int) == 4`
  - 4 x 8 bits = 32 bits
  - each bit represents a power of 2 in the result
  - these bits are literally on/off states of transistors in your computer

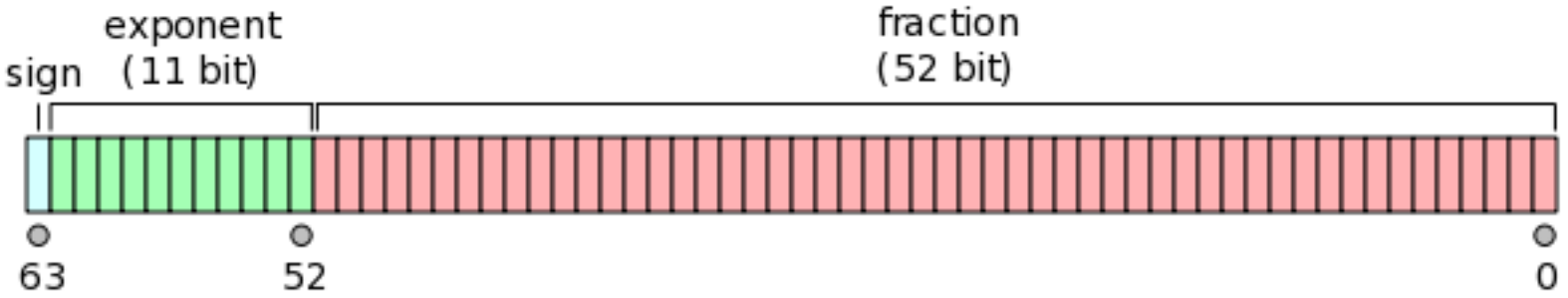
- sign bit (0 positive, 1 negative)
  - most significant bit
- reading to larger addresses indicates less significance
- $2^1 + 2^2 + 2^5 = 38$ (we count from zero in C/C++)
- ….but this arrangement is just for *Big-Endian* processors

IEEE 754 Floating point :$(-1)^s \times c \times b^q$

- `float (binary32 b=2)`
  - sign bit: 1 bit
  - q=exponent width: 8 bits
  - c=significand precision: 24 (23 explicitly stored)



- `double (binary64 b=2)`
  - sign bit: 1 bit
  - q=exponent width: 11 bits
  - c=significand precision: 53 bits (52 explicitly stored)



C

End the suspense! compile and run

```
>g++ prog2.cpp
>./a.out
0 -5 -20
 3 80000024.0 80000026.0  -25.00000
```

```
float f=80000023, e[16];
double d;
```

- First declaration, the compiler sees an integer, and initializes a float using it. did it work ?
  - A question of if there is a combination of significand and exponent that can represent this number with the number of bits available.
- ‘e’ is an array of 16 floats, one after the next.
  - note that the first element is referenced as e[0] (zero-indexing)
- What is ‘d’ at the end of the second line of execution ?

Control Structures: `for` loop

```
int loopI=0;
 for(int j=0; loopI<3; loopI++, j=j-5)
 {
    int result = loopI * j;
 }
```

- `loopI` visible in contained scope
- On entry to loop, `int j=0` executed, then conditional `loopI<3` is checked, if `true`, then loop body is executed, then `loopI++, j=j-5`, then conditional checked……
- When conditional becomes `false`, `result` goes *out of scope* … what does that mean ? What about `j` ?

Conditionals:

- The operators

`<, <=, >, >=, !=, ==, &&, ||, !`

`are used to` define conditional expressions.

  - all binary operations (except !) that return a 0 or a 1 (that's what C calls `true` and `false`

```
if(result == 35 || g < testFunction(3,5))
  {
      if(testFunctionAlternate(g, result<5))
        {
          printf("What a day this was %d\n!", result<5);
        }
      else
        {
          printf("keep trying\n");
        }
   }
```

Pointers and Addresses

- Recall that in the Von Neumann architecture *everything* is *data* in *memory* (cache maintains this *illusion* for the processor)
- Every location in memory has an *address*
- Memory starts at the address 0, and counts forward one byte at a time. Pointers are just integers, giving an index into an “array of memory”.
- Each type has an associated pointer type to go with it, which then becomes it’s own type in C / C++.

```
int  anInteger;
int *anIntegerPointer;
float f;
float *f_ptr;
float **f_pointerToPointer;
char *argv[];     // you've seen this fellow before
void *p;          // what does this mean?
```

Pointers

```
#include <cstdio>
int main(int argc, char* argv[])
{
  int* ptr; // declaring a pointer to an int
  int j = 1;
  int k = 2;
  ptr = &k; //'&' means "take the address of"
  printf("j has the value %d and is stored at %p\n", j, &j);
  printf("j has the value %d and is stored at %llu\n", j, (long long)&j);
  printf("k has the value %d and is stored at %llu\n", k, (long long)&k);
  printf("ptr has the value %p and is stored at %p\n", ptr, &ptr);
  printf("The value of the integer pointed to by ptr is %d\n", *ptr);
  printf("Even the function main can be found in memory %p\n",main);
}
```

- & is the “address of” operator. It will give you the address in memory of any variable or function.
- * is the dereference operator. It means return the value at this address. To return a value, you must know the type – so dereferencing a void pointer will cause an error.

Pointers: program output

```
> /a.out
j has the value 1 and is stored at 0x7fff5fbff37c
j has the value 1 and is stored at 140734799803260
k has the value 2 and is stored at 140734799803256
ptr has the value 0x7fff5fbff378 and is stored at
  0x7fff5fbff370
The value of the integer pointed to by ptr is 2
Even the function main can be found in memory 0x100000d4c
```

- address notation has the leading 0x, and is written in hexidecimal notation
- we can see that the compiler placed j and k right after one another
- What do we think the long long value of `0x7fff5fbff370 is ?`

Memory Management: `new` and `delete`.

The most important usage of pointers is that in conjunction with allocating and deallocating contiguous chunks of memory at run time. In C++, the syntax uses the keywords `new` and `delete`.

```
#include <cstdio.h>
int main(int argc,char *argv[])
{int m;
sscanf(argv[1],"%d",&m);
int *ptr;
ptr = new int[m];
for (int i=0; i < m; i++) { ptr[i] = i;}
long long total = 0;
for (int i=0;i < m; i++)
{total = total + ptr[i]*ptr[i];}
delete [] ptr;
printf("sum of squares of 1 thru %d = %lld \n",m,total);
}
```

Memory Management: `new` and `delete`.

```
ptr = new int[m];
```

This statement allocates a contiguous block of memory of size `sizeof(int)*m`. You can think of this as allocating a 1D array of `int`, with `ptr` giving the address of the first element of the array.

- Address arithmetic: `ptr+i` , where `i` is an `int`, gives the address of an `int` located at address `ptr + i*sizeof(int)`. `ptr++==ptr+1`. We can use an array notation for dereferencing such a location: `ptr[i]==*(ptr + i)`.

Why not just declare `ptr` as an array ?

- When we compiled the program, we did not know the value of `m`.

```
float d[4];// that's allowed.
```

```
float d[m]; // illegal C++
```

- Data allocated with `new` can be returned the system using `delete`.

Memory Management: `new` and `delete`.

The most important use of pointers is in conjunction with allocating and deallocating contiguous chunks of memory at run time. In C++, the syntax uses the keywords `new` and `delete`.

```
int *ptr;  // declares ptr to be a pointer to an integer.
ptr = new int[m];  // allocates a chunk of memory of size m*sizeOf(int) and
                   // and stores the first address of that chunk in ptr.
for (int i=0; i < m; i++) { ptr[i] = i;}  // ptr[i] = *(ptr + i) is of type int .
...
for (int i=0;i < m; i++)
{total = total + ptr[i]*ptr[i];}
delete [] ptr;  // The system memory manager find this address in memory, and
                // labels the chunk of memory of size m*sizeOf(int) starting at ptr
                // as available for use (the memory manger knows what m is).
                // This function has nothing to do with the name ptr, but with the
                // address stored in the variable ptr.
...
}
```

pointer vs. array

```
float a[5];  float* b = new float(5);
for(int i=0; i<5; i++)
{
   a[i] = i;
   b[i] = i;
}
delete[] b;
```

- For most cases array and pointer of a type can be used same way. but not always

```
b++;   // works fine
a++; // compiler error
```

Functions cont. C: a *pass-by-value*

- Like Matlab, C is a pass-by-value, return-by-value language, not a pass by address language like Fortran.
- This might seem very limiting. I can pass back whole arrays of objects from a Matlab function, in C I just get to return one type.
- An example program – finding the maximum value in an array
  - What is really passing-by-value when I send in an array ?

```
float vmax(float v[], int length)
{
  float max = v[0];
  for(int i=1; i<length; i++)
    {
      if(v[i] > max){max = v[i];}
    }
  return max;
}
int main(int argc, char* argv[])
{
  float myArray[5] = {-34.54, 23.4, 15, 88, 2};
  float m = vmax(myArray, 5);
}
```

Declare-before-use also applies to functions.

- Try putting definition of `vmax` after `main`, see what happens.
- This is why `#include <cstdio.h>` at the top of the file (i.e. in global scope).

Pass-by-value, but…..

- Addresses are just another kind of value
- Pointers are the usual data type for addresses
  - But you saw an example of a value being the address of the array
  - You can think of it as a pointer to the first value in the array
- Questions:
  - What would be the effect of changing the value of an element of `v` at the end of the function?
  - What would be the effect of changing the value of `length` at the end of the function?
  - What would be the value of sizeof(v) in the function? What do we actually know about the array if all we passed was its address?
- An example program – what if you also want to return the location of the maximum?

Summary of the C subset

- Programs are compiled, then executed.
- Data types
  - POD. `int, float, double` - arithmetic operations. `char.`
  - Distinction between declaration (naming a variable and its type) and defining (setting its value). Strong static typing -> declare before define.
- Pointers = addresses of PODs are “first-class objects”. Can give them names, and they have their own set of operations on them.
  - Address-of (&var), value-in (*ptr) (latter AKA *dereferencing*).
  - Dynamic allocation of storage. `new` calls a system function to reserve a contiguous chunk of memory; `delete []` tells the system to return the memory to be available for reuse.
- Missing: `struct` (a mechanism for aggregating PODs).

Summary of the C subset

- Functions – call-by-value / return-by-value.
- Loops and conditionals.
- Scope – variables defined inside `{...}` are local, i.e. are no longer defined after the bracket is closed.
  - Functions, loop bodies, conditionals.
  - Variables defined outside `{...}` are accessible from the inner scope.

This is a pretty small set of features – but enough to write Unix-style operating systems.

The Operating System

- Yes, a sidetrack, but this is nice knowledge to have
- What program starts it all ? The **Kernel**
  - The Kernel is the generic name given to the ancestor of all programs
    - Your shell is the great great great great great grandchild *process* of the Kernel
  - Your computer uses firmware and information on disk to get the Kernel running. This is called *booting*.
- A modern operating system is multi-tasking (previously called time-sharing)
  - The Kernel visits it's children and and based on priority and the scheduling protocol picks the next process that gets to continue running.
  - That process is loaded back into the CPU, and allowed to run as if it was the only program on the processor for a period time called the *time-slice* (often 10ms for many systems)
  - The process is put back to sleep and the Kernel takes over again, picking the next process to get to execute.

cont…

- When the main function finishes, Kernel takes the return value, then wakes up and gives it to shell and goes back to sleep itself.
- shell stores the return value in the special shell variable $?
- shell presents the prompt to user and is ready for new input.
- > echo $?
- shell parses the input string. Sees a built-in command ‘echo’, calls it’s built-in function echo with the rest of the string.
- echo parses ‘$?’ and recognizes the special variable for the last function return value. Finds the value of 5, and prints it to STDOUT.

- STDIN and STDOUT are special files under unix.
  - STDIN is a ‘file’ of what you are typing on your keyboard or other inputs.
  - STDOUT is a special file that shows up in a terminal window, or wherever the calling program has redirected this ‘file’