**Midterm Topics**

**Program Organization**

* System layers: C language, libraries, and operating system
* Building an executable: preprocessing, compiling, linking
  + Preprocessor: #include, #define, #ifndef, etc.
* Structure of C/C++ programs: header files, source files
  + Declarations vs definitions
  + Organization and use of header files, including #ifndef guards
  + Faking modularity in C – headers, implementations
  + Internal vs. external linkage; use of static for internal linkage
  + Dependencies and recompilation
  + Make and makefile basics
* Version control basics (git)
  + add, commit, push, etc.
  + What to put in a repo and what to leave out

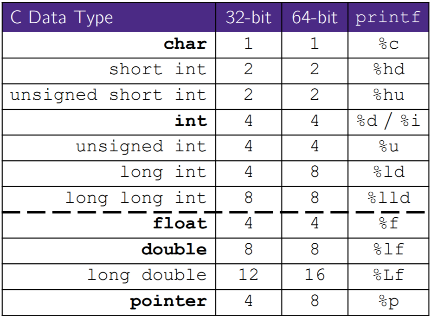
**C Language**

* Data types
  + Casting
  + Pointers (pointer arithmetic, relationship to arrays)
  + String constants, arrays of characters, C string library
  + Structs
  + Extended integer types (e.g. int32\_t, uint64\_t)
  + typedef
* Standard I/O library and streams
  + stdin, stdout, fopen, fread, scanf, printf, etc.
* POSIX libraries - wrappers for systems calls
  + POSIX-layer I/O: open, read, write, etc.
  + Relationship between C standard library, POSIX library functions, and system calls
* Error handling - error codes and errno
* Process address space and memory map (code, static data, heap, stack)
  + Object lifetimes: static, automatic, dynamic
  + Dynamic allocation (malloc, free)
* Function parameters
  + Call by value semantics (including structs, pointers)
  + Arrays as parameters - pointers
  + Using pointers for call-by-reference semantics
  + Function pointers as parameters
* Linked data structures (e.g. linked lists, hash tables)
* Potential bugs (e.g. memory leaks, dangling pointers)

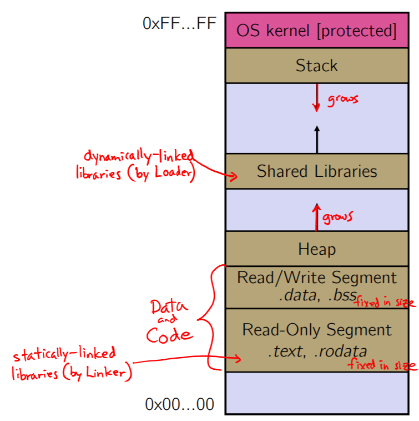
**C++ Language**

* Classes and modularity, namespaces
  + Create and change simple class definitions
  + Constructors and copy constructors, assignments, destructors
* Other basic differences from C
  + Simpler, type-safe stream I/O (cout, cin, << and >>)
  + Type-safe memory management (new, delete, delete[])
  + References – particularly reference parameters
  + More pervasive use of const (const data and parameters, const member functions)
* Templates and libraries
  + Basics of templates - template parameters for classes and functions, when templates are expanded with actual types, templates and header files
  + Basics of STL containers and iterators - particularly vectors

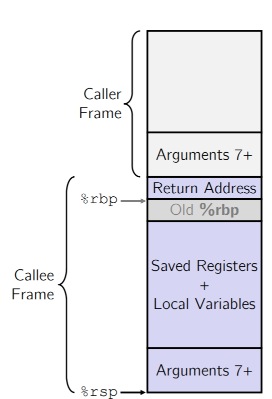
**Lecture Notes.**

**C does not** support objects. **Strings** are null-terminated char arrays and **Structs** are the most object-like feature but are just collections of fields. **Arrays** are contiguous chunks of memory. C is **Call-By-Value.**

**Function Declaration vs. Definition:** Definition is the thing itself, Declaration is description of a thing. Must be exactly one definition of each thing. Declaration needs to appear in all files that use that thing.

**OS and Processes:** The OS lets you run multiple applications at once, time slices each CPU between runnable processes The OS gives each process the illusion of its own private memory. An application runs within an OS “process”.

**Processes and Virtual Memory:** Called the process’ address space. Virtual Memory contains code, data, libraries, stack, etc.

**Loading:** When the OS loads a program it: 1) Creates an address space 2) Inspects the executable file to see what’s in it 3) (Lazily) copies regions of the file into the right place in the address space 4) Does any final linking, relocation, or other needed preparation.

**Memory Management: Local variables on the Stack** - Allocated and freed via calling conventions (push, pop, mov) . **Global and static variables in Data** - Allocated/freed when the process starts/exits. **Dynamically-allocated data on the Heap** - malloc() to request; free() to free, otherwise memory leak.

**The Stack:** Used to store data associated with function calls, compiler-inserted code manages stack frames for you. **Stack frame (x86-64**) includes: Address to return to saved registers - Based on calling conventions Local variables Argument build - Only if > 6 used. Local variables, including arrays, are allocated on the Stack. They “disappear” when the function returns!

**Pointers:** Data types or variables that store addresses, it points to somewhere in the process’ virtual address space. For example, &foo produces the virtual address of foo. Generic definition: **type\* name;** or **type \*name;** Dereference a pointer using the unary \* operator. We use them for several things in C, such as: Simulating “pass-by-reference” Using function arguments as return values (also known as “output parameters​”). Avoiding copying huge structs when passing arguments into functions.

**Pointer Arithmetic:** We can do addition and subtraction to pointers, with a catch. Arithmetic on pointers is scaled to the size of the type being pointed to (in bytes). For example, ptr + 1 would actually increase the value of ptr by 4 since for a 32-bit integer. void\* is a generic pointer (i.e. a placeholder). Pointer arithmetic is scaled by sizeof(\*p)

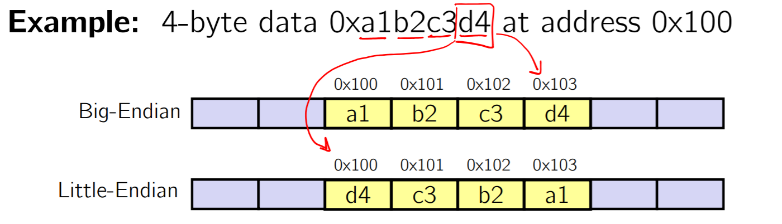
**Output Parameters:** As the name suggests, this refers to a parameter that we use to store the output of a function. These are very common in C and you will see a lot of libraries use these.

**Address Space Layout Randomization:** Linux uses address space layout randomization (ASLR) for added security.

**Arrays:** Definition: **type name[size].** Allocates **size\*sizeof(type)** **bytes** of contiguous memory. Normal usage is a compile-time constant for size (e.g. int scores [175] ;). Initially, array values are “garbage”. Size of an array is not stored anywhere – array does not know its own size! The sizeof(array) only works in variable scope of array definition. Recent versions of C allow for variable-length arrays which is uncommon and can be considered bad practice.

**Multi-dimensional Arrays: type name[rows][cols]** = {{values},…,{values}}; Still allocates a single, contiguous chunk of memory, C is row-major.

**Parameters: reference vs. value: T**here are two fundamental parameter-passing schemes in programming languages. **Call-by-value** - Parameter is a local variable initialized when the function is called and gets a copy of the calling argument; manipulating the parameter only changes copy, not the calling argument. In C, Java, C++ primitives. **Call-by-reference**: Parameter is an alias for the supplied argument; manipulating the parameter manipulates the calling argument. C++ arrays and references.

**Endianness:** Memory is byte-addressed, so endianness determines what ordering that multi-byte data gets read and stored in memory.

**Big-endian**: Least significant byte has highest address.

**Little-endian:** Least significant byte has lowest address.

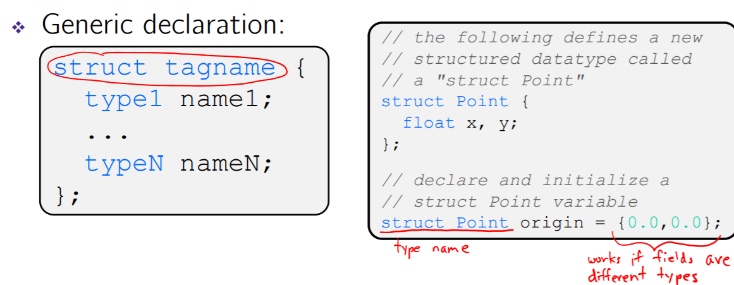
**Faking Call-By-Reference in C:** Can use pointers to approximate call-by-reference. Callee still receives a copy of the pointer (i.e. call-by-value), but it can modify something in the caller’s scope by dereferencing the pointer parameter.

**Function Pointers:** Can use pointers that store addresses of functions! Generic format: returnType (\* name)(type1, …, typeN). Looks like a function prototype with extra \* in front of name.

**Memory Allocation:** Global variables are statically allocated, local variables are automatically allocated. Local variables are allocated when function is called and deallocated when function returns.

**Dynamically Allocation:**  **malloc** allocates a block of memory of the requested size. Returns a pointer to the first byte of that memory annd returns NULL if the memory allocation failed! General usage: var = (type\*) malloc(size in bytes). **calloc()** is like malloc, but also zeros out the block of memory helpful for shaking out bugs and is slightly slower; preferred for non-performance-critical code. malloc and calloc are found in stdlib.h. General Usage: var = (type\*) calloc(num, bytes per element). **free()** deallocates the memory pointed-to by the pointer. Pointer must point to the first byte of heap-allocated memory (i.e. something previously returned by malloc or calloc). The freed memory becomes eligible for future allocation. Pointer is unaffected by call to free. Usage: free(pointer).

**The Heap:** The Heap is a large pool of unused memory that is used for dynamically-allocated data. *malloc* allocates chunks of data in the Heap; *free* deallocates those chunks.

**Memory Leak**: A memory leak occurs when code fails to deallocate dynamically-allocated memory that is no longer used. e.g. forget to *free* malloc-ed block, lose/change pointer to malloc-ed block.

**Structured Data**: A struct is a C datatype that contains a set of fields. Like a Java class, but with no methods or constructors. Useful for defining new structured types of data, acts similarly to primitive variables. Use “.” to refer to a field in a struct. Use “->” to refer to a field from a struct pointer.

**Copy by Assignment**: You can assign the value of a struct from a struct of the same type – this copies the entire contents!

**typedef:** Generic format: **typedef type name.** typedef is a keyword in the C and C++ programming languages. The purpose of typedef is to assign alternative names to existing types, most often those whose standard declaration is cumbersome, potentially confusing, or likely to vary from one implementation to another**.** It allows you to define new data type names/synonyms. Both type and name are usable and refer to the same type; Be careful with pointers – \* before name is part of type! You can malloc and free structs, just like other data type. *sizeof* (typename) is particularly helpful here.

**Pass Copy of Struct or Pointer? Value passed**: passing a pointer is cheaper and takes less space unless struct is small. **Field access:** indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize. For small structs, passing a copy of the struct can be faster and often preferred; for large structs use pointers.

**Multi-File C Programs (modules):** A module is a self-contained piece of an overall program and has externally visible functions that customers can invoke. Module also has externally visible typedefs, and perhaps global variables, that customers can use and may have internal functions, typedefs, or global variables that customers should not look at. The module’s interface is its set of public functions, typedefs, and global variables.

**Modularity:** It is defined as the degree to which components of a system can be separated and recombined. Modules can be developed independently and can be re-used in different projects.

**C Header Files:** Header: A C file whose only purpose is to be #include’d. Generally, has a filename .h extension. Holds the variables, types, and function prototype declarations that make up the interface to a module. Every name.c is intended to be a module that has a name.h The name.h declares the interface to that module. Other modules can use name by #include-ing name.h and they should assume as little as possible about the implementation in name.c.

**C Module Conventions: Most C projects adhere to the following rules:**

♣ .h files only contain declarations, never definitions.

♣ .c files never contain prototype declarations for functions that are intended to be exported through the module interface. Those function prototype declarations belong in the .h file.

♣ NEVER #include a .c file – only #include .h files.

♣ #include all of headers you reference, even if another header (accidentally or not) includes some of them.

♣ Any .c file with an associated .h file should be able to be compiled into a .o file • The .c file should include the .h file; the compiler will check definitions and declarations.

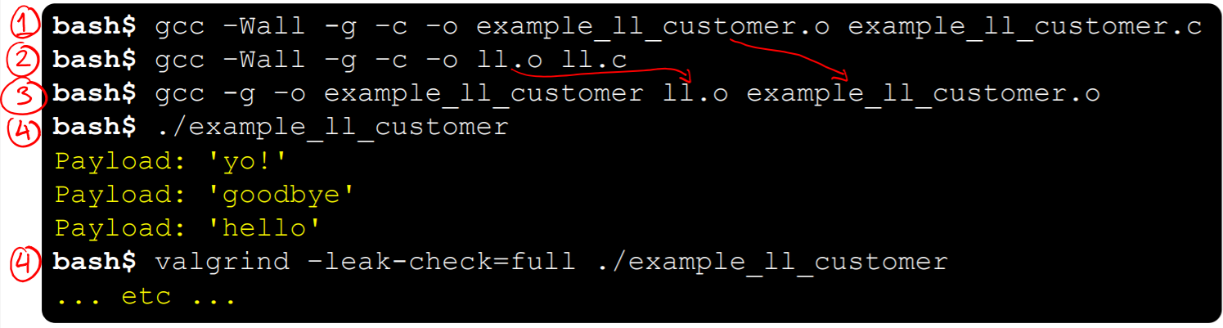
**Compiling the Program:** It has four parts:

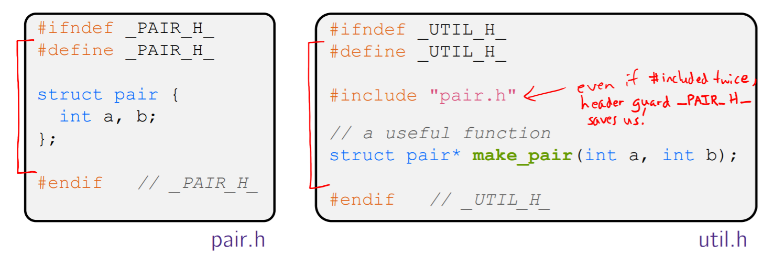
♣ 1/2) Compile example\_ll\_customer.c into an object file

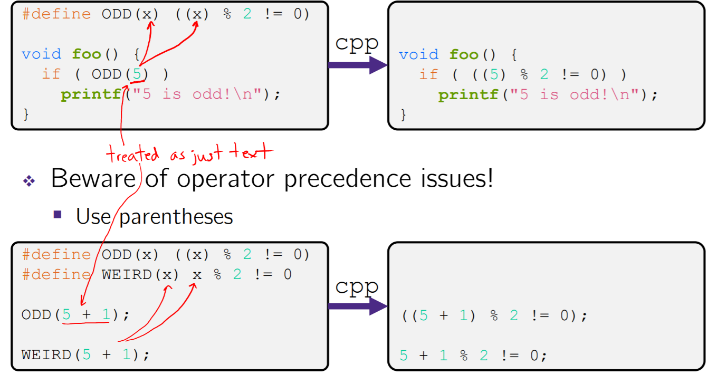
♣ 2/1) Compile ll.c into an object file

♣ 3) Link both object files into an executable

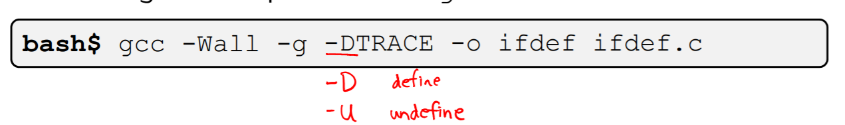
♣ 4) Test, Debug, Rinse, Repeat.



**Header Guards:** A commonly-used C Preprocessor trick to deal with this. Uses macro definition (#define) in combination with conditional compilation (#ifndef and #endif)

**Macros:** You can pass arguments to macros.

**Conditional Compilation:** You can change what gets compiled with #ifdef and #ifndef.

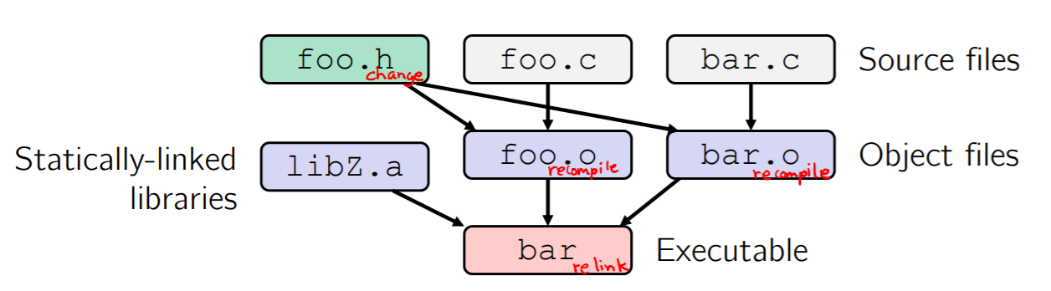
**Defining Symbols:** Besides #defines in the code, preprocessor values can be given as part of the gcc command.

**External Linkage:** *extern* makes a declaration of something externally visible. Every global (variables and functions) is extern by default. Unless you add the static specifier, if some other module uses the same name, you’ll end up with a collision!

**Internal Linkage:** *static* (in the global context) restricts a definition to visibility within that file. It’s good practice to use static to “defend” your global variables. C has a different use for the word “static”: to create a persistent local variable. The storage for that variable is allocated when the program loads, in either the .data or .bss segment.

**Make:** *make* is a classic program for controlling what gets (re)compiled and how. make has tons of fancy features, but only two basic ideas: 1) Scripts for executing commands 2) Dependencies for avoiding unnecessary work.

**Recompilation Management:** The “theory” behind avoiding unnecessary compilation is a “dependency dag” (directed, acyclic graph). To create a target , you need sources s\_1, s\_2, s\_3 and a command that directly or indirectly uses the sources. It is newer than every source (file-modification times), assume there is no reason to rebuild it. Recursive building: if some source s\_i is itself a target for some other sources, see if it needs to be rebuilt… Cycles “make no sense”!



* Compiling a .c creates a .o – the .o depends on the .c and all included files (.h, recursively/transitively).
* An archive (library, .a) depends on included .o files.
* Creating an executable (“linking”) depends on .o files and archives.
* If one .c file changes, just need to recreate one .o file, maybe a library, and re-link.
* If a .h file changes, may need to rebuild more.

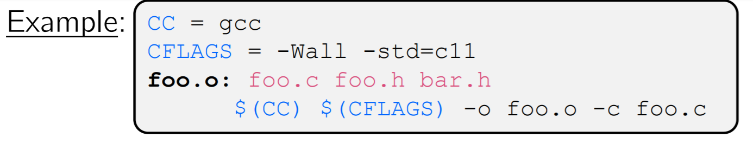
**make Basics:** A makefile contains a bunch of triples:

♣ Colon after target is required

♣ Command lines must start with a TAB, NOT SPACES

♣ Multiple commands for same target are executed in order

• Can split commands over multiple lines by ending lines with ‘\’

♣ You can define variables in a Makefile:

All values are strings of text, no “types”

Variable names are case-sensitive and can’t contain

‘:’, ‘#’, ‘ =’, or whitespace.

Easy to change things (especially in multiple commands)

Can also specify on the command line (CFLAGS=-g)

♣ clean is a convention . Removes generated files to “start over” from just the source

It’s “funny” because the target doesn’t exist and there are no sources, but it works because:

• The target doesn’t exist, so it must be “remade” by running the command

• These “phony” targets have several uses, such as “all”.

♣ Special variables:

$@ for target name

$^ for all sources

$< for left-most source

**Using make:** Defaults:-

♣ If no -f specified, use a file named Makefile

♣ If no target specified, will use the first one in the file

♣ Will interpret commands in your default shell

• Set SHELL variable in makefile to ensure.

Target execution: -

♣ Check each source in the source list:

• If the source is a target in the Makefile, then process it recursively

• If some source does not exist, then error

• If any source is newer than the target (or target does not exist), run command (presumably to update the target)

**File I/O.** These functions are part of *glibc* on Linux.They are implemented using Linux system calls. C’s *stdio* defines the notion of a stream: a way of reading or writing a sequence of characters to and from a device.

♣ Can be either text or binary; Linux does not distinguish

♣ Is buffered by default; *libc* reads ahead of you

♣ Three are provided by default: *stdin, stdout, stderr*

• You can open additional streams to read and write to files

♣ Streams have the type FILE\*, which is defined in *stdio.*

**Buffering:** By default, stdio turns on buffering for streams:

* + Data written by fwrite() is copied into a buffer allocated by stdio inside your process’ address space.
  + As some point, the buffer will be “drained” into the destination:

• When you explicitly call fflush() on the stream

• When the buffer size is exceeded (often 1024 or 4096 bytes)

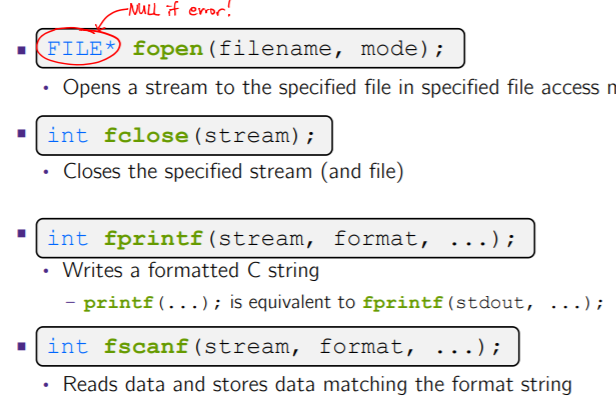
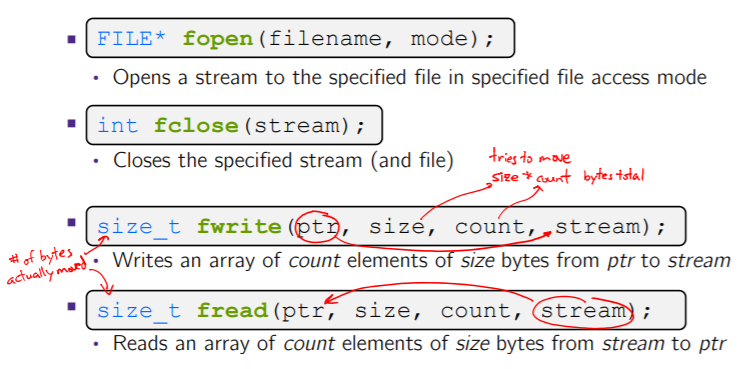
• For stdout to console, when a newline is written (“line buffered”)

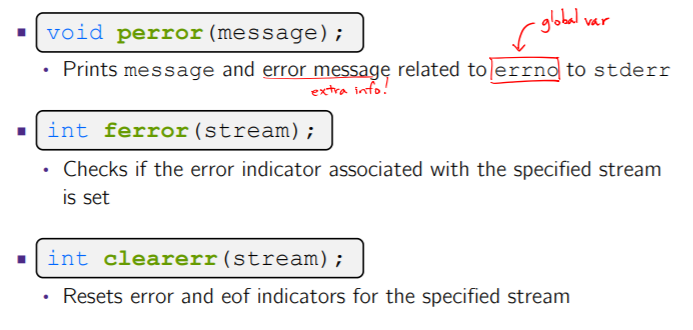
• When you call fclose() on the stream

• When your process exits gracefully (exit() or return from main())

**C Stream Functions:**

Some stream functions (complete list in stdio):



**Error Checking/Handling:** Some error functions (complete list in stdio):

**Buffering Issues:** Performance implications:

♣ Data is copied into the stdio buffer

• Consumes CPU cycles and memory bandwidth

• Can potentially slow down high-performance applications, like a web server or database (“zero-copy”)

Solutions:

♣Turn off buffering with setbuf(stream, NULL)

Unfortunately, this may also cause performance problems

• e.g. if your program does many small fwrite()s, each of which will now trigger a system call into the Linux kernel

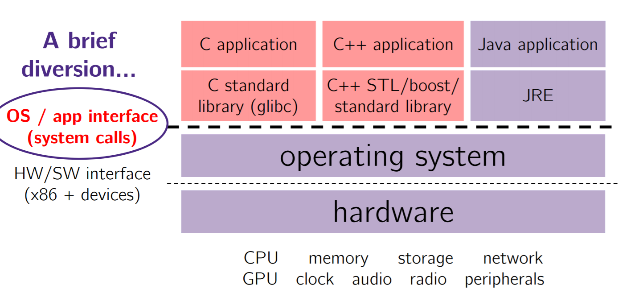
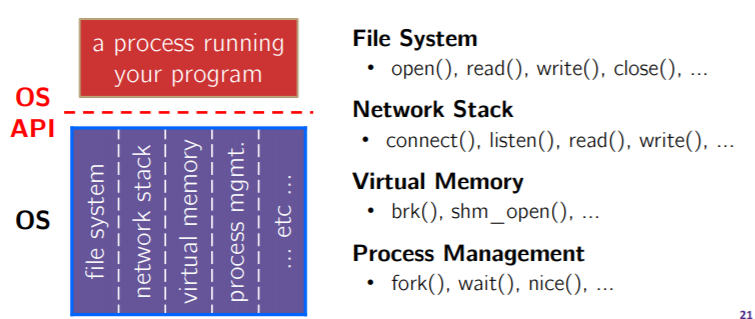
♣ Use a different set of system calls

♣ POSIX (OS layer) provides open(), read(), write(), close(), etc.

• No buffering is done at the user level

• The OS caches disk reads and writes in the FS buffer cache

• Disk controllers have caches too!



**System Calls:**

**What’s an OS?**  A Software that**:**

♣ Directly interacts with the hardware

• OS is trusted to do so; user-level programs are not

• OS must be ported to new hardware; user-level programs are portable

♣ Manages (allocates, schedules, protects) hardware resources

• Decides which programs can access which files, memory locations, pixels on the screen, etc. and when

♣ Abstracts away messy hardware devices

• Provides high-level, convenient, portable abstractions (e.g. files, disk blocks)

*The OS is the “layer below”*

♣ A module that your program can call (with system calls)

♣ Provides a powerful OS API – POSIX, Windows, etc.

**OS - Protection System:**

OS isolates process from each other

♣ But permits controlled sharing between them

• Through shared name spaces (e.g. FS names)

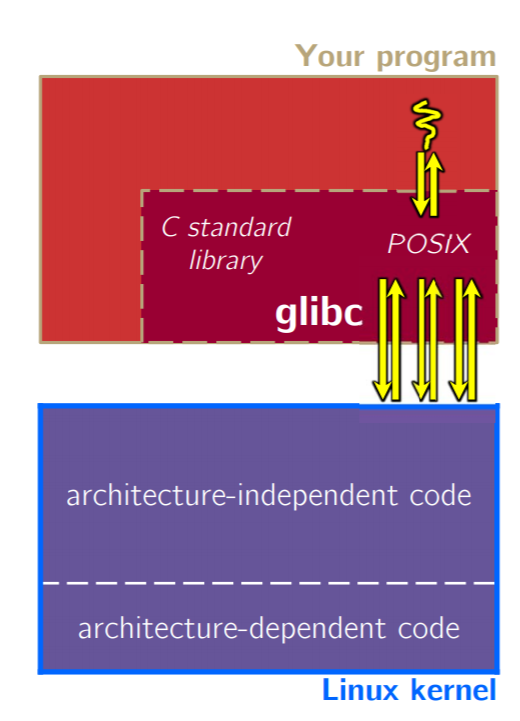
♣ OS isolates itself from processes

• Must prevent processes from accessing the hardware directly

♣ OS can access the hardware

• User-level processes run with the CPU in unprivileged mode

• The OS runs with the CPU in privileged mode

 • User-level processes invoke system calls to safely enter the OS.

**Details on x86/Linux\*:** Its thread of execution can be in one of several places:

• In your program’s code

• In glibc, a shared library containing the C standard library, POSIX, support, and more

• In the Linux architectureindependent code

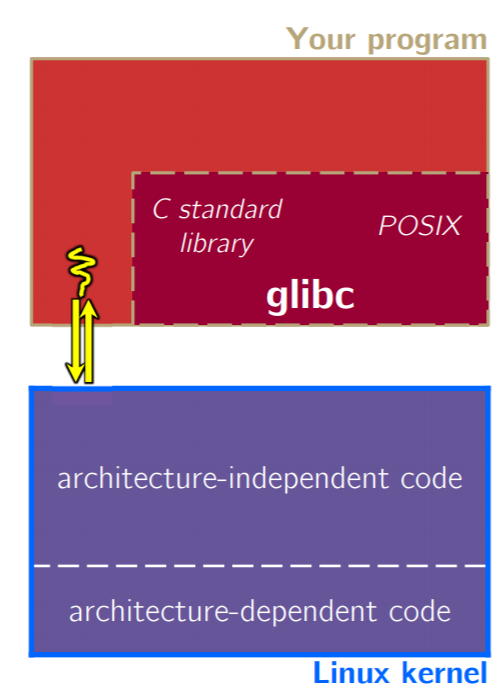
• In Linux x86-64 code.

Some routines may be handled by glibc, but they in turn invoke Linux system calls

♣ e.g. POSIX wrappers around Linux syscalls

• POSIX readdir() invokes the underlying Linux readdir()

♣ e.g. C stdio functions that read and write from files.



Your program can choose to directly invoke Linux system calls as well

♣ Nothing is forcing you to link with glibc and use it

♣ But relying on directly-invoked Linux system calls may make your program less portable across UNIX varieties.

Your program can access many layers of APIs:

♣ C standard library

♣ POSIX compatibility API

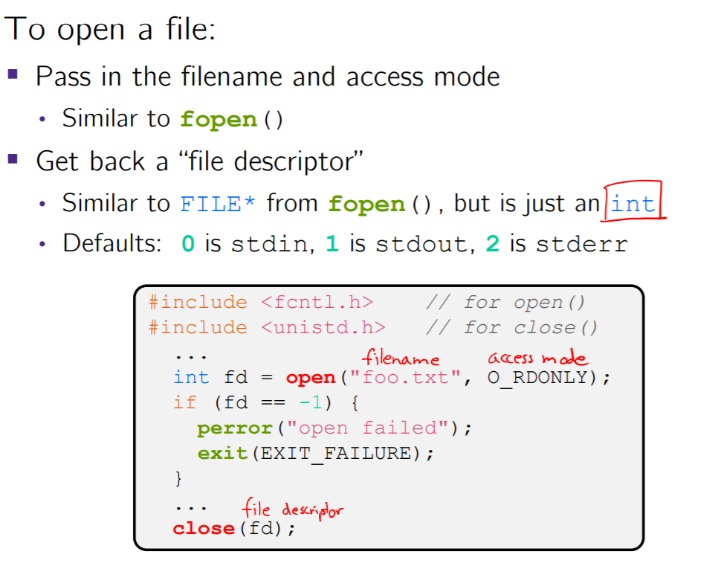
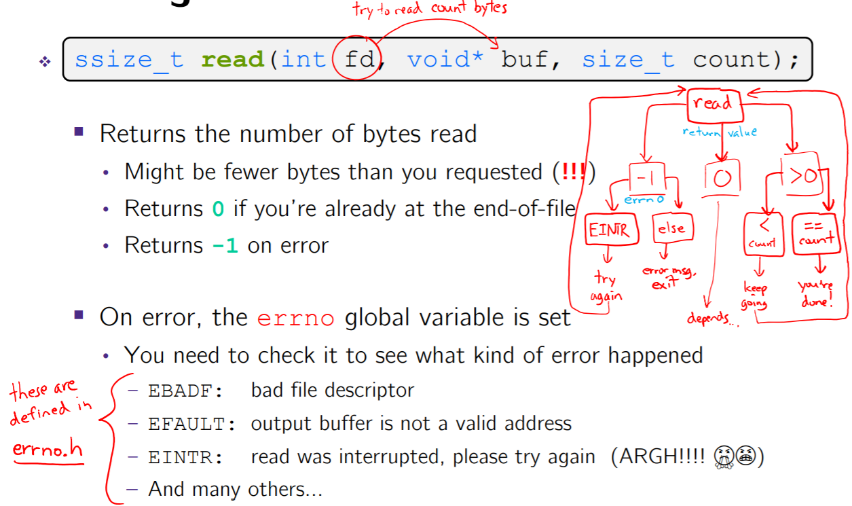
♣ Underlying OS system calls

**Lower-Level File Access:** Most UNIX-en support a common set of lower-level file access APIs: POSIX – Portable Operating System Interface. open(), read(), write(), close(), lseek().

• Similar in spirit to their f\*() counterparts from C std lib

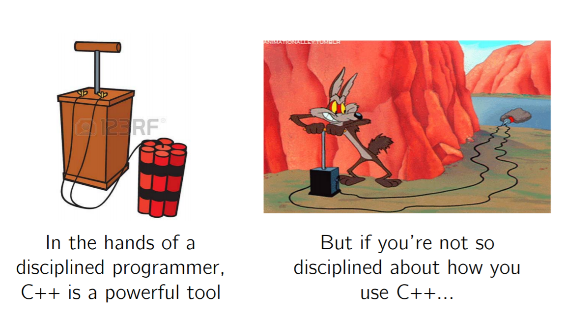
• Lower-level and unbuffered compared to their counterparts

• Also, less convenient.

**open()/close(): Reading from a File:**

**Other Low-Level Functions:** <http://www.cs.cmu.edu/~guna/15-123S11/Lectures/Lecture24.pdf>



**C vs C++ comparison:**

In **C** we had to work hard to mimic encapsulation, abstraction

♣ Encapsulation: hiding implementation details

• Used header file conventions and the “static” specifier to separate private functions from public functions

• Cast structures to void\* to hide implementation-specific details (generalize)

♣ Abstraction: associating behavior with encapsulated state

• Function that operate on a LinkedList were not really tied to the linked list structure

• We passed a linked list to a function, rather than invoking a method on a linked list instance.

In **C ++** a major addition is support for classes and objects!

♣ Classes

• Public, private, and protected methods and instance variables

• (multiple!) inheritance

♣ Polymorphism

• Static polymorphism: multiple functions or methods with the same name, but different argument types (overloading) – Works for all functions, not just class members

• Dynamic (subtype) polymorphism: derived classes can override methods of parents, and methods will be dispatched correctly.

In **C** we had to emulate generic data structures

♣ Generic linked list using void\* payload

♣ Pass function pointers to generalize different “methods” for data structures

• Comparisons, deallocation, pickling up state, etc.

**C++** supports templates to facilitate generic data types.

♣ Parametric polymorphism – same idea as Java generics, but different in details, particularly implementation

♣ To declare that x is a vector of ints: *vector x;*

♣ To declare that x is a vector of floats: *vector x;*

♣ To declare that x is a vector of (vectors of floats): *vector<vector> x;*

In **C** We had to be careful about namespace collisions.

♣ C distinguishes between external and internal linkage

• Use static to prevent a name from being visible outside a source file (as close as C gets to “private”)

• Otherwise, name is global and visible everywhere

♣ We used naming conventions to help avoid collisions in the global namespace

• e.g. LLIteratorNext vs. HTIteratorNext, etc.

**C++** permits a module to define its own namespace!

♣ The linked list module could define an “LL” namespace while the hash table module could define an “HT” namespace.

♣ Both modules could define an Iterator class

• One would be globally named LL::Iterator and the other would be globally named HT::Iterator

♣ Classes also allow duplicate names without collisions.

• Namespaces group and isolate names in collections of classes and other “global” things (somewhat like Java packages)

**C** does not provide any standard data structures

♣ We had to implement our own linked list and hash table

♣ As a C programmer, you often reinvent the wheel… poorly

• Maybe if you’re clever you’ll use somebody else’s libraries

• But C’s lack of abstraction, encapsulation, and generics means you’ll probably end up tweak them or tweak your code to use them.

**The C++** standard library is huge!

♣ Generic containers: bitset, queue, list, associative array (including hash table), deque, set, stack, and vector

• And iterators for most of these

♣ A string class: hides the implementation of strings

♣ Streams: allows you to stream data to and from objects, consoles, files, strings, and so on.

**In C** error handling is a pain

♣ Must define error codes and return them

♣ Customers must understand error code conventions and need to constantly test return values ♣ e.g. if a() calls b(), which calls c()

• a depends on b to propagate an error in c back to it.

C++ supports exceptions!

♣ try / throw / catch

♣ If used with discipline, can simplify error processing

• But, if used carelessly, can complicate memory management

• Consider: a() calls b(), which calls c() – If c() throws an exception that b() doesn’t catch, you might not get a chance to clean up resources allocated inside b()

♣ But much C++ code still needs to work with C & old C++ libraries, so still uses return codes, exit(), etc.

**Some Tasks Still Hurt in C++:**

Memory management

♣ C++ has no garbage collector

• You have to manage memory allocation and deallocation and track ownership of memory • It’s still possible to have leaks, double frees, and so on

♣ But there are some things that help • “Smart pointers” – Classes that encapsulate pointers and track reference counts – Deallocate memory when the reference count goes to zero.

C++ doesn’t guarantee type or memory safety

♣ You can still:

• Forcibly cast pointers between incompatible types

• Walk off the end of an array and smash memory

• Have dangling pointers

• Conjure up a pointer to an arbitrary address of your choosing.

**Other C++ Features:**

Operator overloading

♣ Your class can define methods for handling “+”, “->”, etc.

Object constructors, destructors

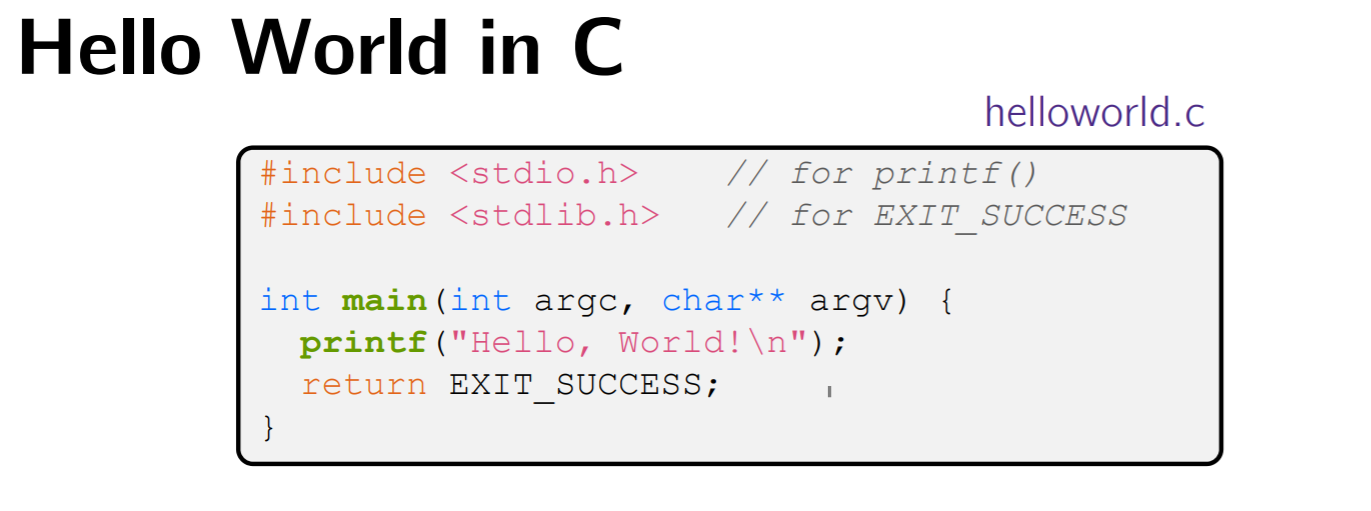
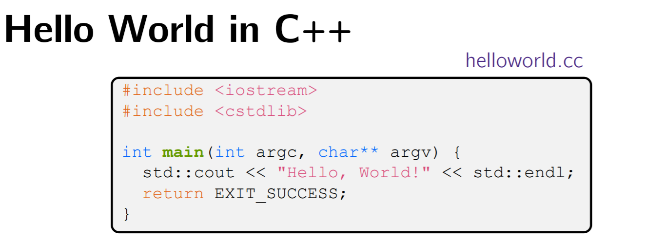
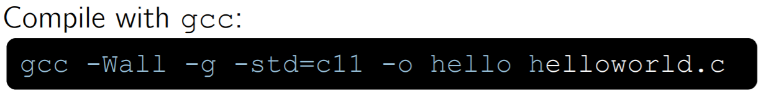
♣ Particularly handy for stack-allocated objects

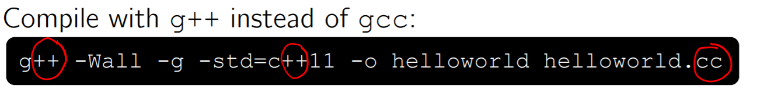
Reference types

♣ Truly pass-by-reference instead of always pass-by-value

Advanced Object Orientedness

♣ Multiple inheritance, virtual base classes, dynamic dispatch.





**C++ Basics:**

C is (roughly) a subset of C++

♣ You can still use printf – but bad style now!

♣ Can mix C and C++ idioms if needed to work with existing code but avoid mixing if you can. Use C++(11)

♣ iostream is part of the C++ standard library

♣ You don’t include “.h” when you include C++ standard library headers

• But you do for local headers (e.g. #include "ll.h")

♣ iostream declares stream object instances in the “std” namespace

• e.g. std::cin, std::cout, std::cerr

*cstdlib* is the C standard library’s stdlib.h

♣ Nearly all C standard library functions are available to you.

*std::cout* is the “cout” object instance declared by iostream, living within the “std” namespace.

♣ C++’s name for stdout

♣ std:cout is an object of class ostream

• <http://www.cplusplus.com/reference/ostream/ostream/>

♣ Used to format and write output to the console

♣ The entire standard library is in the namespace std.

**C++** distinguishes between objects and primitive types

♣ These include the familiar ones from C: char, short, int, long, float, double, etc.

♣ C++ also defines bool as a primitive type.

*ostream* has many different methods to handle <<

♣ The functions differ in the type of the right-hand side (RHS) of <<

♣ e.g. if you do std::cout << "foo"; , then C++ invokes cout’s function to handle << with RHS char\*

The ostream class’ member functions that handle << return a reference to themselves.

When std::cout << "Hello, World!"; is evaluated: A member function of the std::cout object is invoked. It buffers the string "Hello, World!" for the console and it returns a reference to std::cout.

Next, another member function on *std::cout* is invoked to handle << with RHS std::endl ♣ std::endl is a pointer to a “manipulator” function • This manipulator function writes newline ('\n') to the ostream it is invoked on and then flushes the ostream’s buffer • This enforces that something is printed to the console at this point.

C++’s standard library has a std::string class . Include the string header to use it. Seems to be automatically included in iostream on CSE Linux environment (C++11).

C++ allows you to truly pass-by-reference

♣ Client passes in an argument with normal syntax

• Function uses reference parameters with normal syntax

• Modifying a reference parameter modifies the caller’s argument!

See: <https://courses.cs.washington.edu/courses/cse333/18sp/lectures/10/CSE333-L10-c++_refs_const_18sp-ink.pdf>

**References:** A reference is an alias for another variable

♣ Alias: another name that is bound to the aliased variable

• Mutating a reference is mutating the aliased variable

♣ Introduced in C++ as part of the language.

**const in C++:** *const*: this cannot be changed/mutated

♣ Used much more in C++ than in C

♣ Signal of intent to compiler; meaningless at hardware level

• Results in compile-time errors.

**Pointers** can change data in two different contexts:

1) You can change the value of the pointer

2) You can change the thing the pointer points to (via dereference)

**const** can be used to prevent either/both behaviors!

♣ const next to pointer name means you can’t change the value of the pointer

♣ const next to data type pointed to means you can’t use this pointer to change the thing being pointed to

♣ Tip: read variable declaration from right-to-left.

A const parameter cannot be mutated inside the function.

Therefore, it does not matter if the argument can be mutated or not.

A non-const parameter may be mutated inside the function

It would be BAD if you passed it a const variable.

**const Tip:**

Use const reference parameters for input values.

♣ Particularly for large values (no copying)

Use pointers for output parameters

♣ Input parameters first, then output parameters last.

**When to use References:** Google C++ style guide suggests:

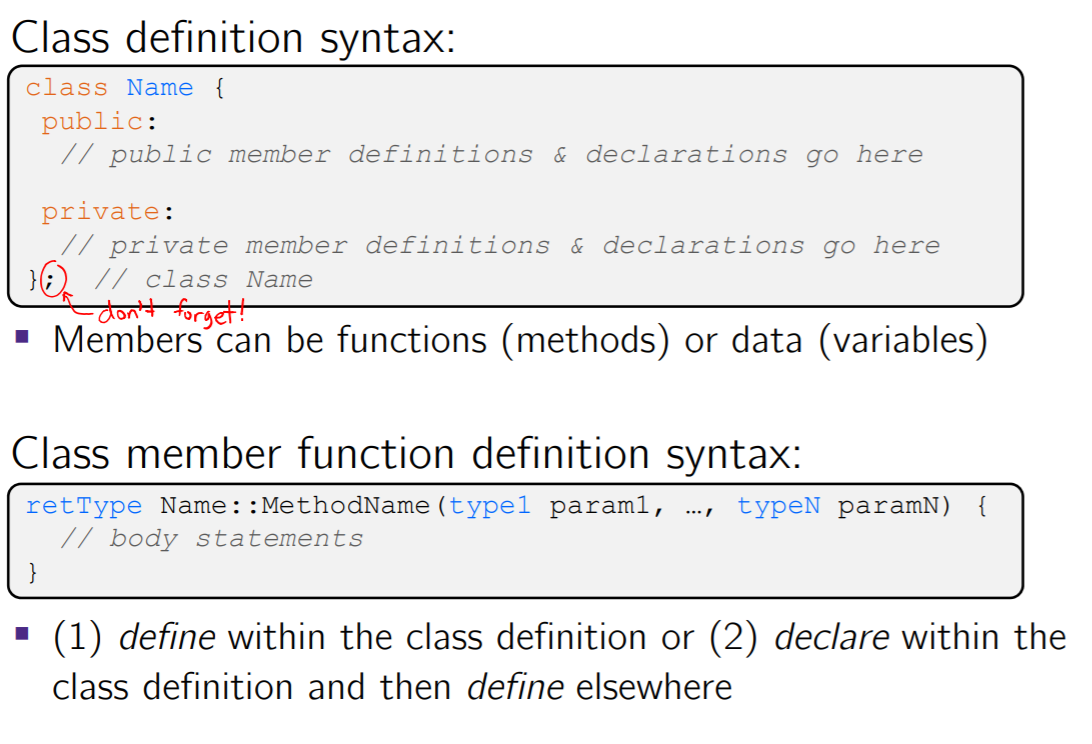
♣ Input parameters:

• Either use values (for primitive types like int or small structs/objects)

• Or use const references (for complex struct/object instances)

♣ Output parameters:

• Use const pointers – Unchangeable pointers referencing changeable data.

**Classes in C++:**

♣ Class definition is part of interface and should go in .h file

• Private members still must be included in definition (!)

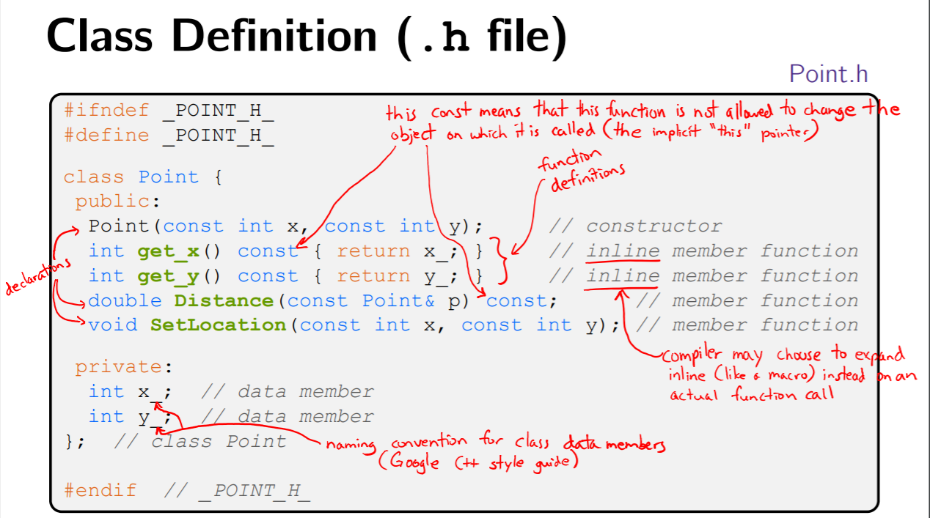
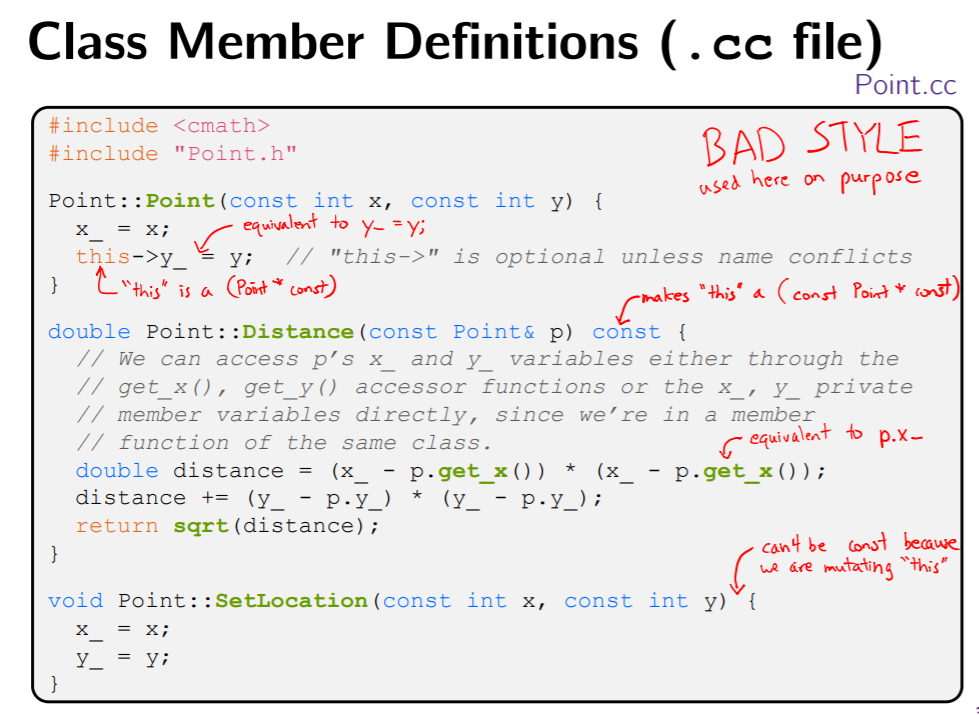
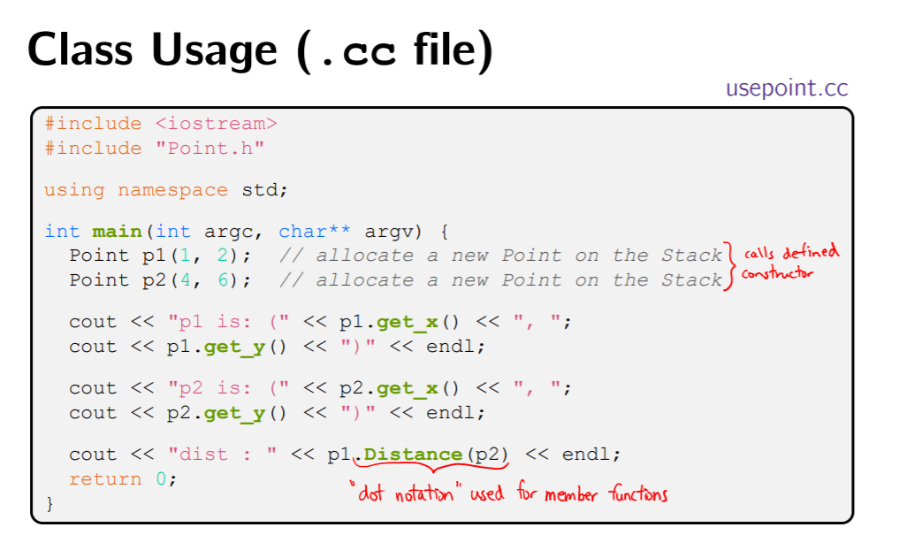
♣ In the interest of hiding details, usually separate member function definitions into companion .cc file

• Common exception: setter and getter methods

♣ These files can also include non-member functions that use the class

♣ Unlike Java, you can name files anything you want

Typically Name.cc and Name.h for class Name.



**struct vs. class:**

In C, a struct can only contain data fields

♣ No methods and all fields are always accessible ϖ In C++, struct and class are (nearly) the same!

♣ Both can have methods and member visibility (public/private/protected)

♣ Minor difference: members are default public in a struct and default private in a class

Common style convention:

♣ Use struct for simple bundles of data

♣ Use class for abstractions with data + functions.

**Constructors:** Example **Point::Point(const int x, const int y);**

A constructor (ctor) initializes a newly-instantiated object .

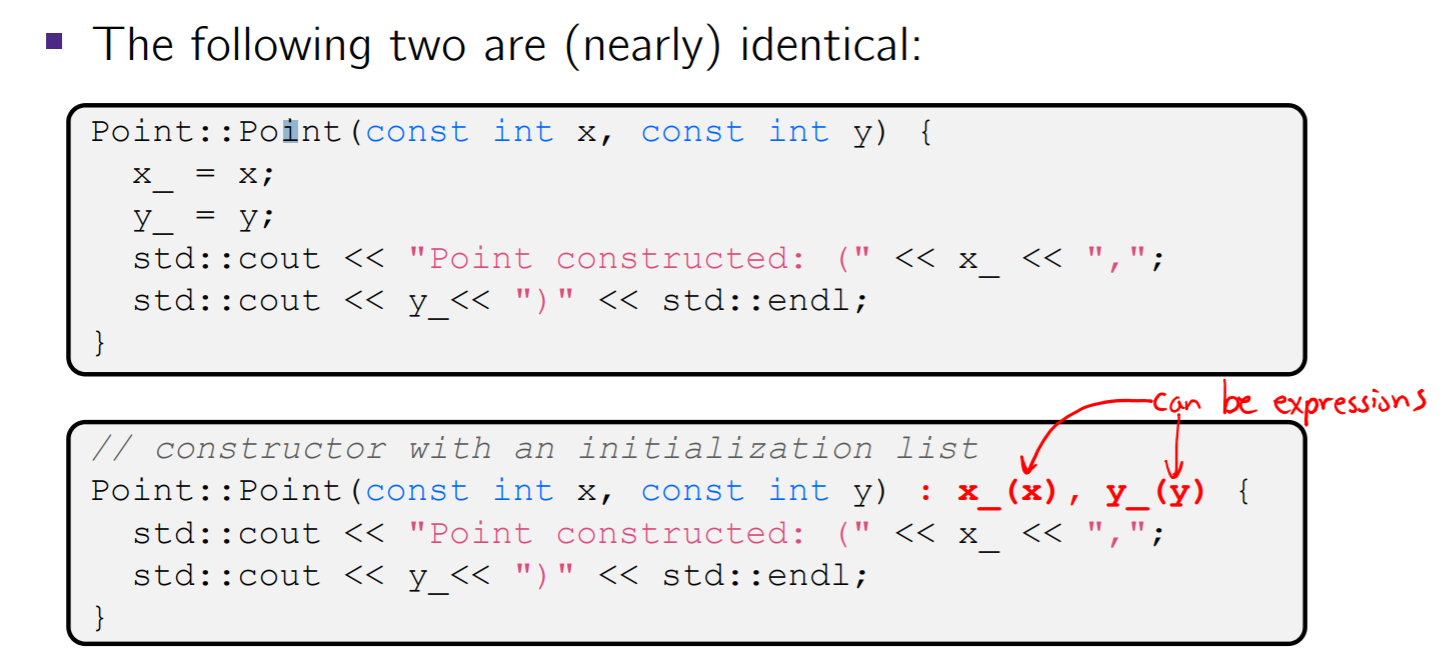
♣ A class can have multiple ctors that differ in parameters

• Which one is invoked depends on how the object is instantiated.

C++ will automatically create a synthesized default constructor if you have no user-defined constructors.

**Initialization Lists:**

C++ lets you optionally declare an initialization list as part of your constructor definition which initializes fields according to parameters in the list

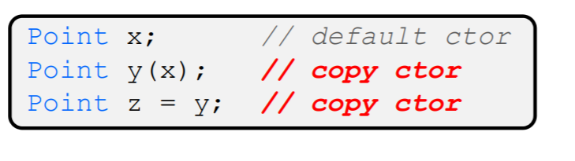


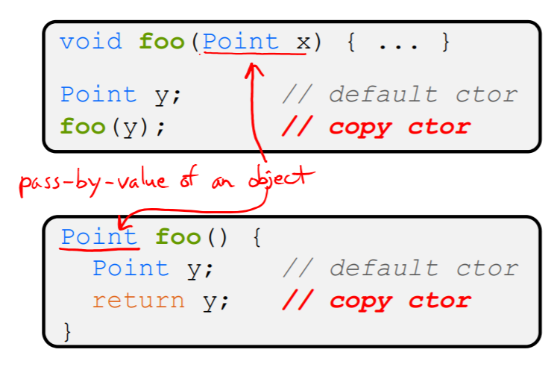
Data members are initialized in the order they are defined, not by the initialization list ordering

• Data members that don’t appear in the initialization list are default initialized/constructed before body is executed. **Initialization preferred to assignment to avoid extra steps.**

**Copy Constructors:**

C++ has the notion of a copy constructor (cctor) which is used to create a new object as a copy of an existing object

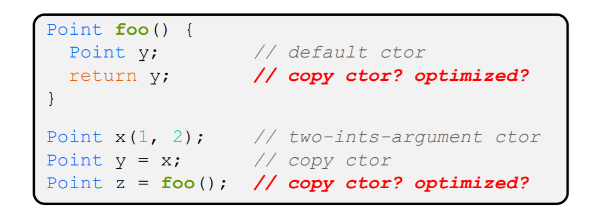
The copy constructor is invoked if:

♣ You initialize an object from another object of the same type:

♣ You pass a non-reference object as a parameter to a function:

♣ You return a non-reference object from a function:

♣ The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies



♣ Sometimes you might not see a constructor get invoked when you might expect it

If you don’t define your own copy constructor, C++ will synthesize one for you

♣ It will do a shallow copy of all of the fields (i.e. member variables) of your class

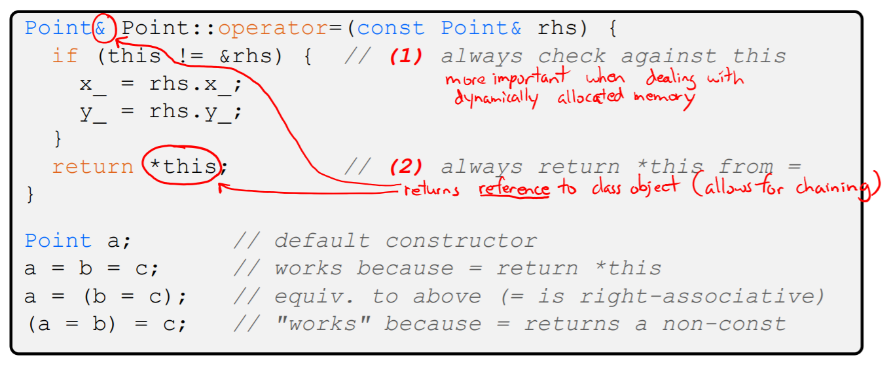
♣ Sometimes the right thing; sometimes the wrong thing.

**Assignment != Construction.**

“ =” is the assignment operator

♣ Assigns values to an existing, already constructed object.

You can choose to overload the “=” operator. But there are some rules you should follow:



f you don’t overload the assignment operator, C++ will synthesize one for you

♣ It will do a shallow copy of all of the fields (i.e. member variables) of your class

♣ Sometimes the right thing; sometimes the wrong thing

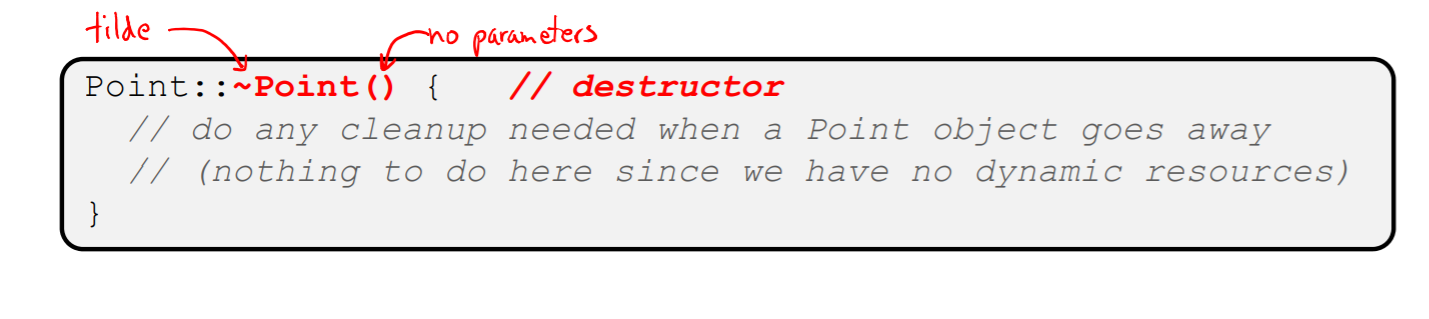
**Destructors:** C++ has the notion of a destructor (dtor)

♣ Invoked automatically when a class instance is deleted, goes out of scope, etc. (even via exceptions or other causes!)

♣ Place to put your cleanup code – free any dynamic storage or other resources owned by the object

♣ Standard C++ idiom for managing dynamic resources

• Slogan: “Resource Acquisition Is Initialization” (RAII)



**Rule of Three**

♣ If you define any of:

* + - 1. Destructor
      2. ) Copy Constructor
      3. Assignment (operator=)

♣ Then you should normally define all three.

If possible, disable the copy constructor and assignment operator by declaring as private and not defining them. If you disable them, then you should instead probably have an explicit “CopyFrom” function.

**Class Encapsulation.**

**Access Control:** Access modifiers for members:

♣ public: accessible to all parts of the program

♣ private: accessible to the member functions of the class

• Private to class, not object instances

♣ protected: accessible to the member functions of the class and any derived classes.

♣ Access modifiers apply to all members that follow until another access modifier is reached

♣ If no access modifier specified, struct members default to public and class members default to private.

**Nonmember Functions:**

“Nonmember functions” are just normal functions that happen to use our class

♣ Called like a regular function instead of as a member of a class object instance

• This gets a little weird when we talk about operators…

♣ These do not have access to the class’ private members.

♣ Useful nonmember functions often included as part of interface

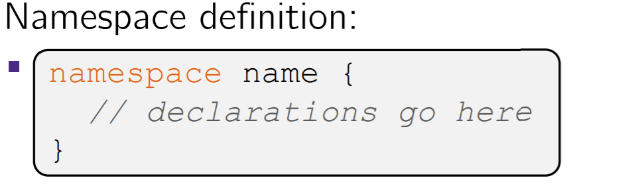
Declaration goes in header file, but outside of class definition.

**friend Nonmember Functions:**

A class can give a nonmember function (or class) access to its nonpublic members by declaring it as a friend within its definition

♣ Access modifiers do not apply; function is not a member

♣ friend functions are unnecessary if your class includes “getter” public functions.



**Namespaces:** Each namespace is a separate scope. Useful for avoiding symbol collisions!

♣ Creates a new namespace name if it did not exist, otherwise adds to the existing namespace (!)

• This means that namespaces can discontiguous

♣ Definitions can appear outside of the namespace definition

**Classes vs. Namespaces:**

They look very similar, but classes are not namespaces:

♣ There are no instances/objects of a namespace; a namespace is just a group of logically-related members.

♣ To access a member of a namespace, you must use the fully qualified name (i.e. nsp\_name::member)

• Unless you are using that namespace

• You only used the fully qualified name of a class member when you are defining it outside of the scope of the class definition.

**C++11 nullptr:** C and C++ have long used NULL as a pointer value that references nothing.

C++11 introduced a new literal for this: nullptr.

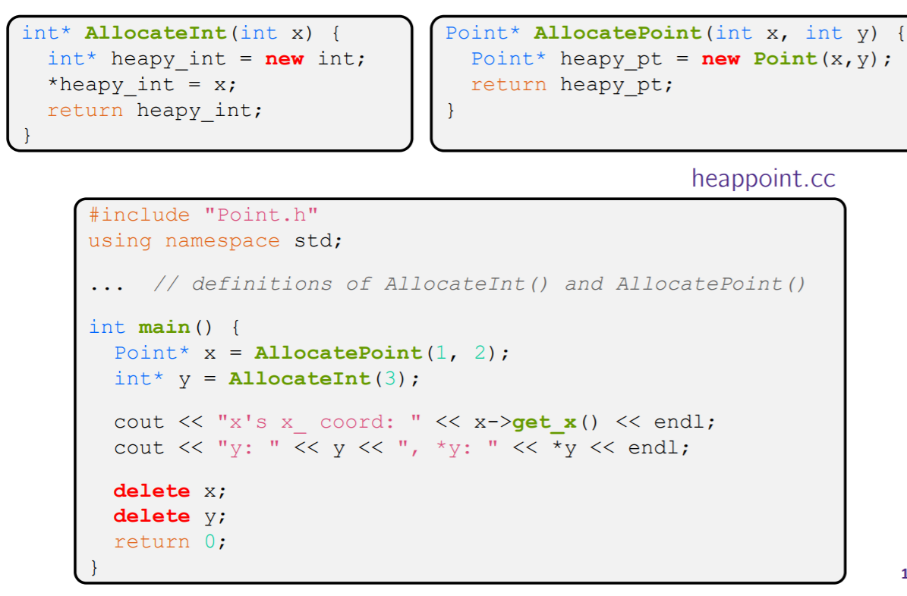
♣ Interchangeable with NULL for all practical purposes, but it has type T\* for any/every T, and is not an integer value

• Avoids funny edge cases (see C++ references for details)

• Still can convert to/from integer 0 for tests, assignment, etc.

♣ Advice: prefer nullptr in C++11 code • Though NULL will also be around for a long, long time.

**Using the Heap:**

To allocate on the heap using C++, you use the new keyword instead of malloc() from stdlib.h

♣ You can use new to allocate an object (e.g. new Point)

♣ You can use new to allocate a primitive type (e.g. new int).

To deallocate a heap-allocated object or primitive, use the delete keyword instead of free() from stdlib.h.

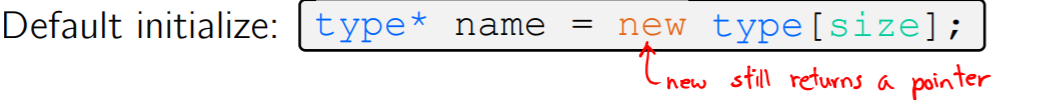
Don’t mix and match!

• Never free() something allocated with new

• Never delete something allocated with malloc()

• Careful if you’re using a legacy C code library or module in C++

**Dynamically Allocated Arrays:** To dynamically allocate an array:

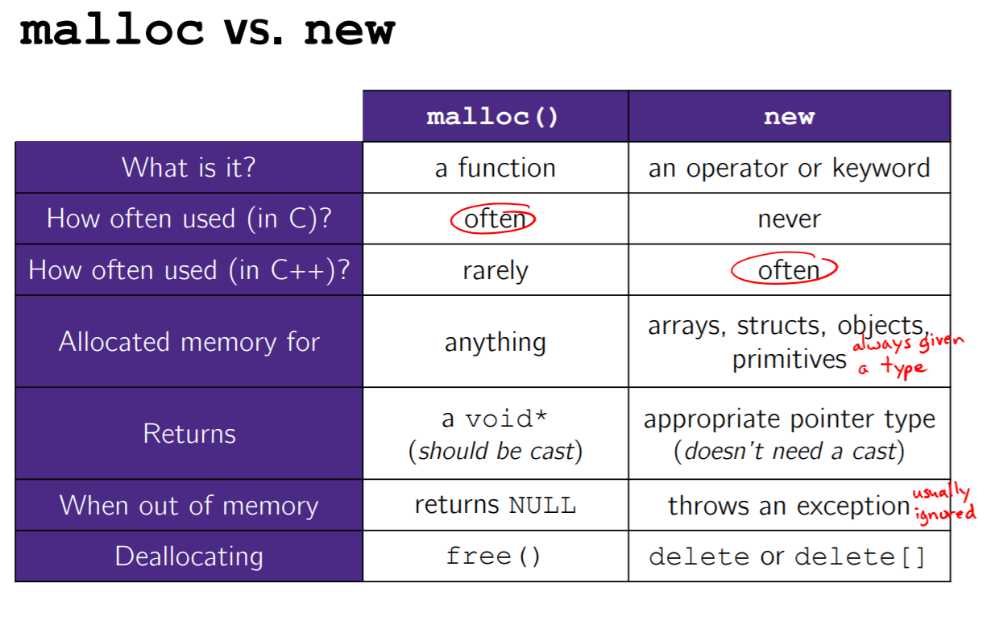


To dynamically deallocate an array:

The compiler probably won’t catch this, though (!)

because it can’t tell if it was allocated with new type[size]; or new type;

Results in undefined behavior.



**Templates:** C++ has the notion of templates

♣ A function or class that accepts a type as a parameter.

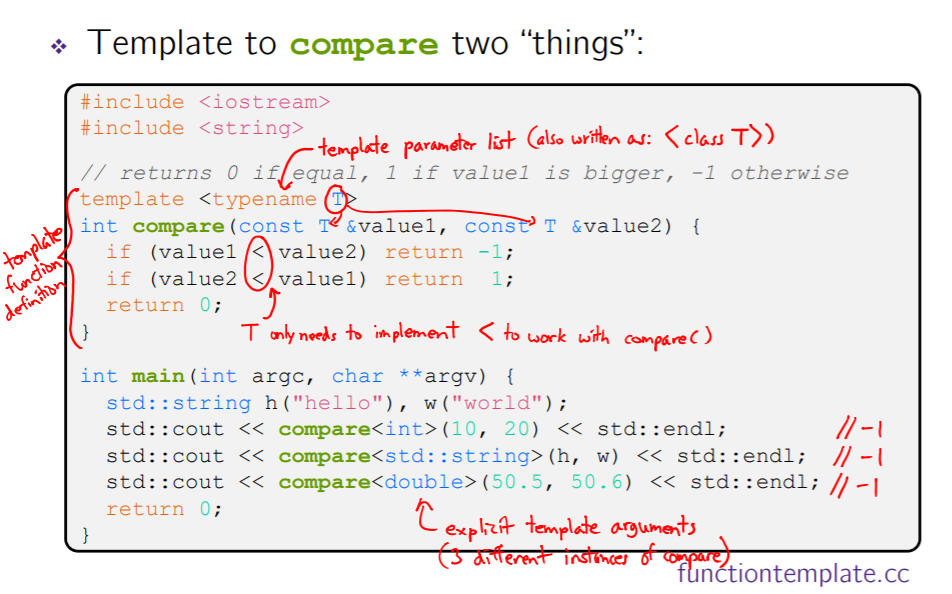
• You define the function or class once in a type-agnostic way

• When you invoke the function or instantiate the class, you specify (one or more) types or values as arguments to it

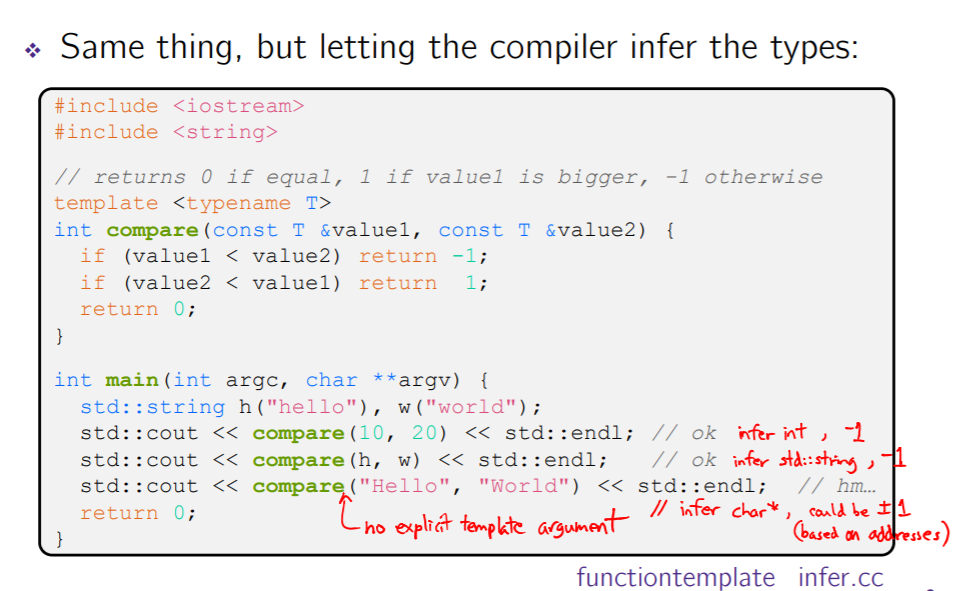
♣ At compile-time, the compiler will generate the “specialized” code from your template using the types you provided.

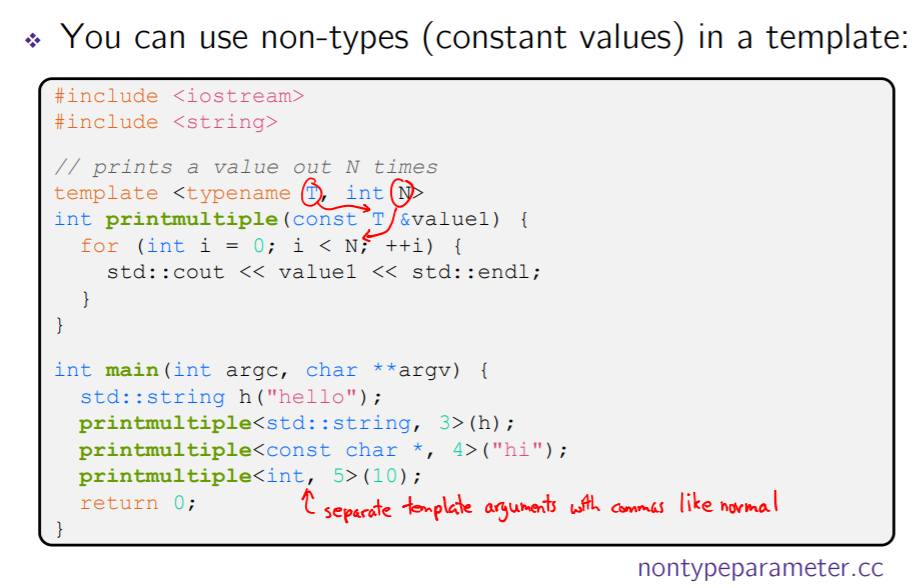
• Your template definition is NOT code

• Code is only generated if you use your template.



**Compiler Inference:**



**Non-Type Templates:**

The compiler doesn’t generate any code when it sees the template function

♣ It doesn’t know what code to generate yet, since it doesn’t know what types are involved.

When the compiler sees the function being used, then it understands what types are involved

♣ It generates the instantiation of the template and compiles it (kind of like macro expansion)

• The compiler generates template instantiations for each type used as a template parameter.

**Class Templates:** Templating is useful for classes as well! Imagine we want a class that holds a pair of things that we can:

♣ Set the value of the first thing

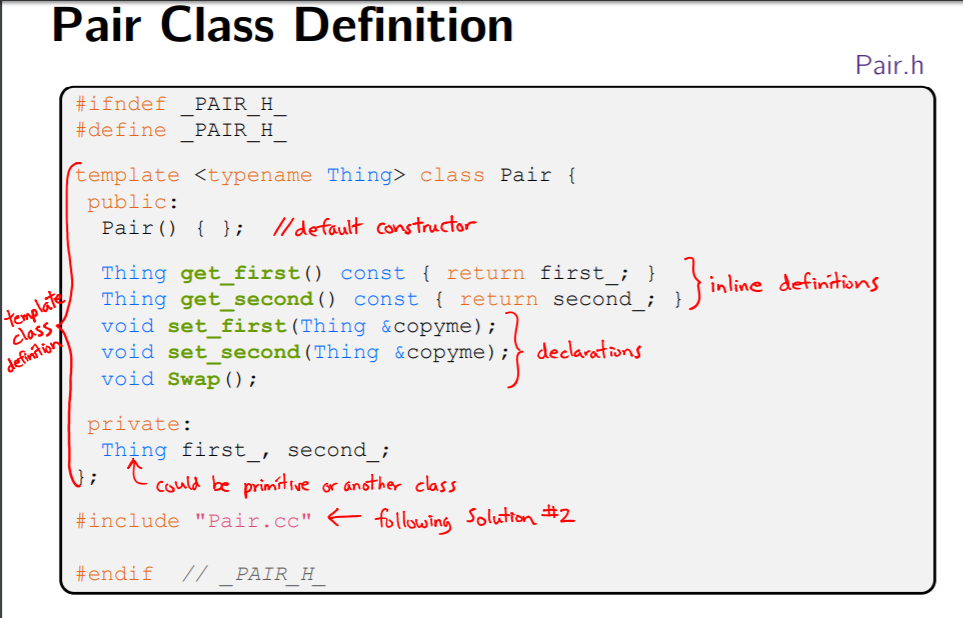
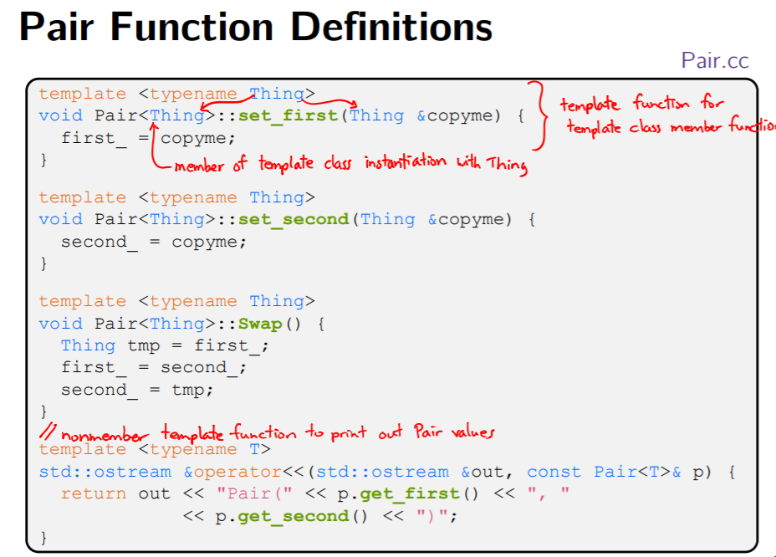
♣ Set the value of the second thing

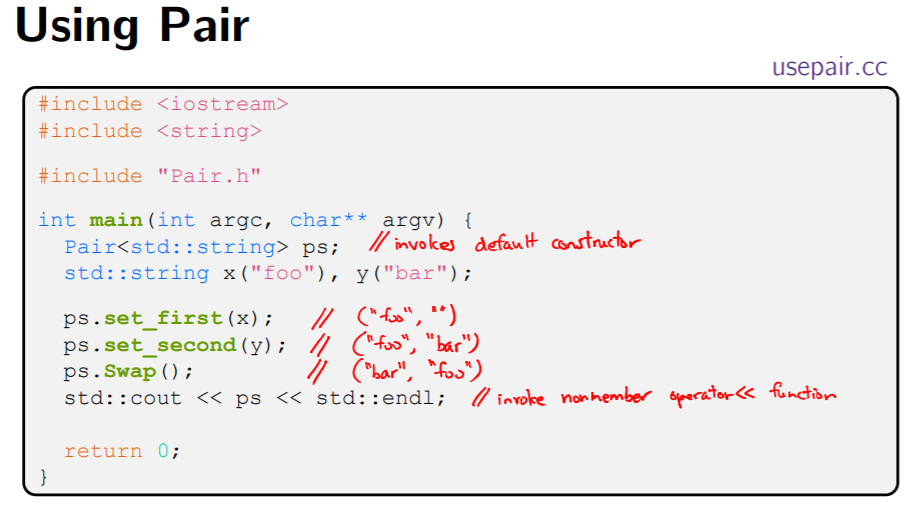
♣ Get the value of the first thing

♣ Get the value of the second thing

♣ Swap the values of the things

♣ Print the pair of things.





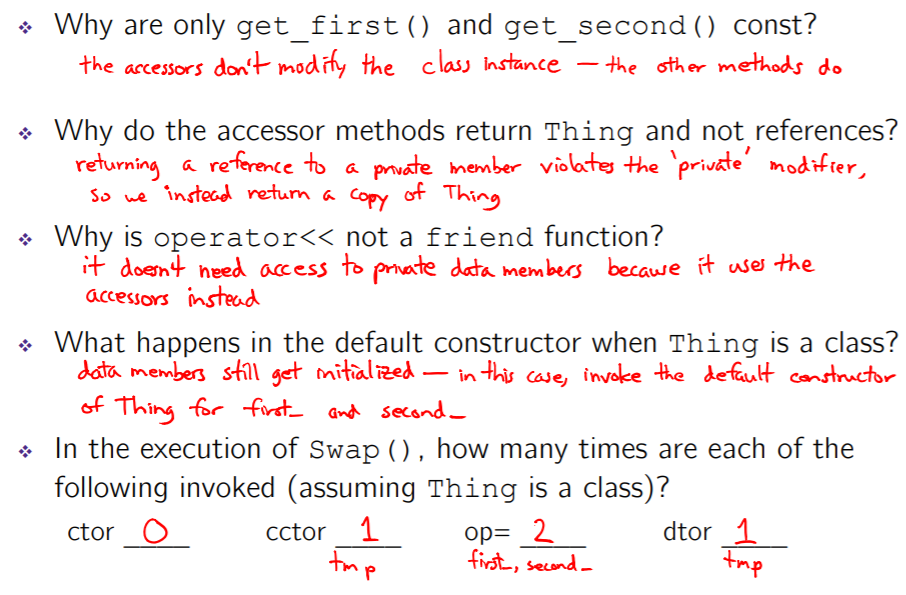
Thing is replaced with template argument when class is instantiated

♣ The class template parameter name is in scope of the template class definition and can be freely used there.

♣ Class template member functions are template functions with template parameters that match those of the class template

• These member functions must be defined as template function outside of the class template definition – The template parameter name does not need to match that used in the template class definition, but really should

♣ Only template methods that are actually called in your program are instantiated



**STL Containers:**

A container is an object that stores (in memory) a collection of other objects (elements). Implemented as class templates, so hugely flexible.

Several different classes of container

♣ Sequence containers (vector, deque, list) - these index numerically

♣ Associative containers (set, map, multiset, multimap, bitset) – these index by key

♣ Differ in algorithmic cost and supported operations.

Drawbacks:

STL containers store by value, not by reference

♣ When you insert an object, the container makes a copy

♣ If the container needs to rearrange objects, it makes copies

• e.g. if you sort a vector, it will make many, many copies

• e.g. if you insert into a map, that may trigger several copies

♣ What if you don’t want this (disabled copy constructor or copying is expensive)?

• You can insert a wrapper object with a pointer to the object –“smart pointers”.

**The STL vector:** A generic, dynamically resizable array.

♣ <http://www.cplusplus.com/reference/stl/vector/vector/>

♣ Elements are store in contiguous memory locations

• Elements can be accessed using pointer arithmetic if you’d like to

• Random access is O(1) time

♣ Adding/removing from the end is cheap (constant time) ♣ Inserting/deleting from the middle or start is expensive (linear time).

**The STL iterator:** Each container class has an associated iterator class used to iterate through elements of the container.

♣ <http://www.cplusplus.com/reference/std/iterator/>

♣ Iterator range is from begin up to end

• end is one past the last container element!

♣ Some container iterators support more operations than others • All can be incremented (++), copied, copy-constructed

• Some can be dereferenced on RHS (e.g. x = \*it;)

• Some can be dereferenced on LHS (e.g. \*it = x;)

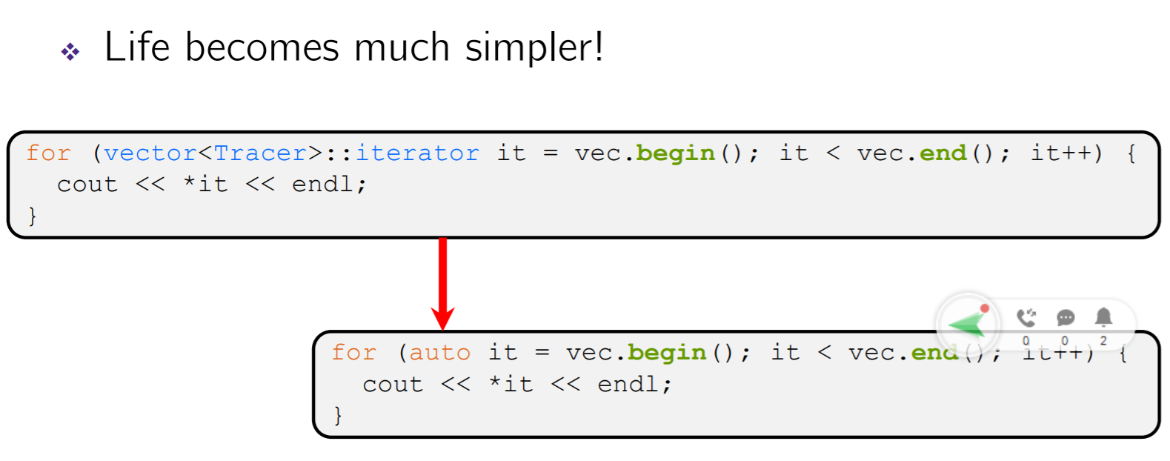
• Some can be decremented (--)

• Some support random access ([], +, -, +=, -=, operators)

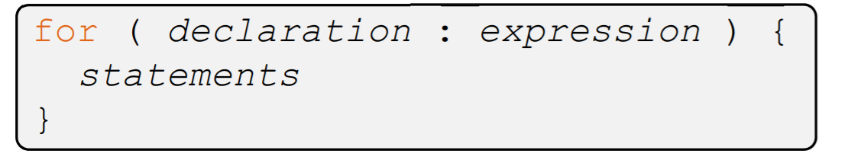
**Type Inference (C++11):**

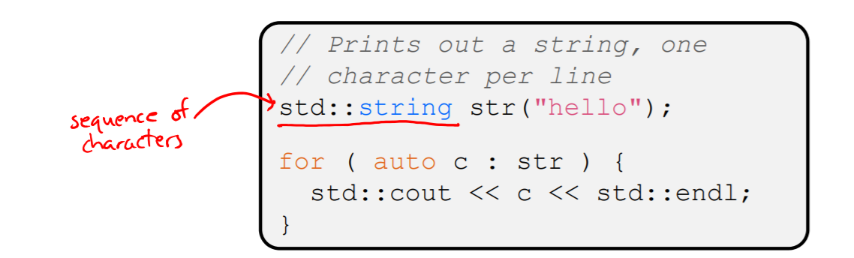
The auto keyword can be used to infer types

♣ Simplifies your life if, for example, functions return complicated types

♣ The expression using auto must contain explicit initialization for it to work

**Range for Statement (C++11):** Syntactic sugar that emulates Java’s foreach.

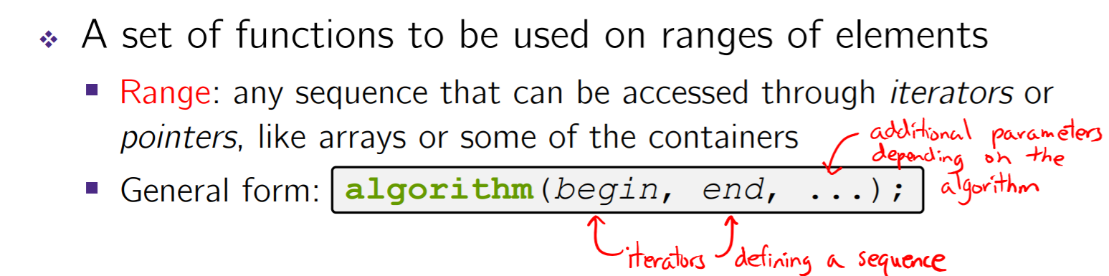




*declaration* defines loop variable.

*expression* is an object representing a sequence

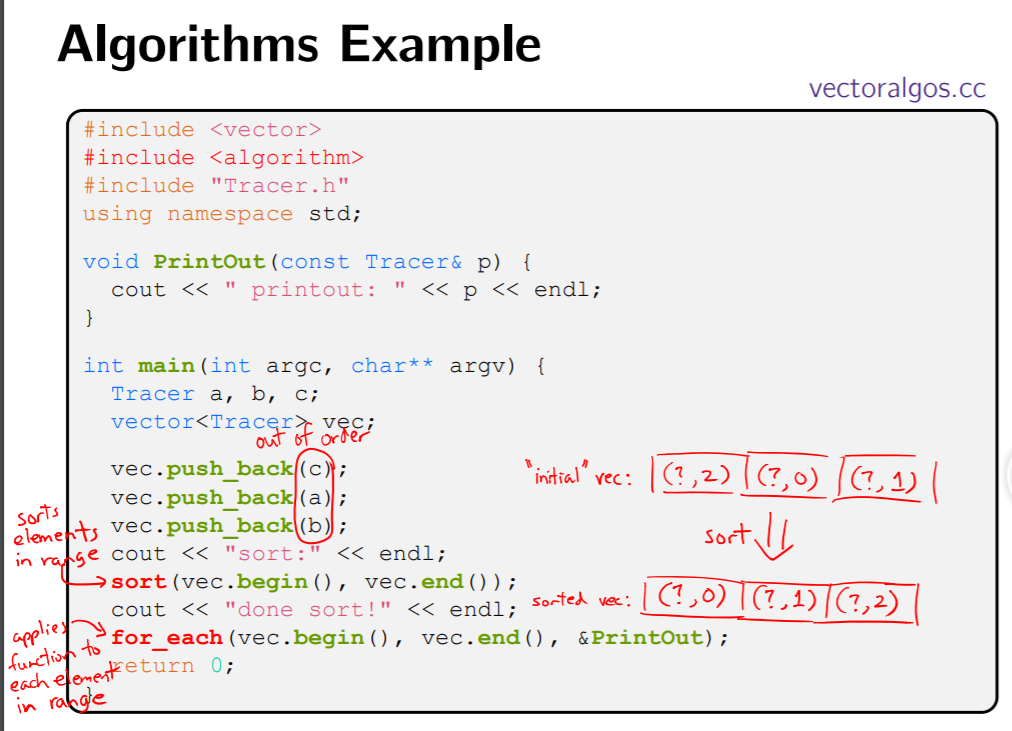
• Strings, initializer lists, arrays with an explicit length defined, STL containers that support iterators.

**STL Algorithms**

♣ Make use of elements’ copy ctor, =, ==, !=, <

♣ Some do not modify elements • e.g. find, count, for\_each, min\_element, binary\_search

♣ Some do modify elements • e.g. sort, transform, copy, swap



**References Revisited:** Refer to [**https://courses.cs.washington.edu/courses/cse333/18sp/lectures/17/CSE333-L17-c++\_review\_18sp-ink.pdf**](https://courses.cs.washington.edu/courses/cse333/18sp/lectures/17/CSE333-L17-c++_review_18sp-ink.pdf) **(Ignore Smart Pointers for the Midterm.)**

**Pointers vs References in C++**

**(Source:** [**https://www.geeksforgeeks.org/pointers-vs-references-cpp/**](https://www.geeksforgeeks.org/pointers-vs-references-cpp/)**)**

C and C++ support pointers which is different from most of the other programming languages. Other languages including C++, Java, Python, Ruby, Perl and PHP support references.

On the surface, both references and pointers are very similar, both are used to have one variable provide access to another. With both providing lots of same capabilities, it’s often unclear what is different between these different mechanisms. In this article, I will try to illustrate the differences between pointers and references.

Pointers: A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with \* operator to access the memory location it points to.

References : A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer is also implemented by storing the address of an object.

A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e the compiler will apply the \* operator for you.

**Differences** :

1. **Reassignment:** A pointer can be re-assigned. This property is useful for implementation of data structures like linked list, tree, etc. See the following examples:

int x = 5;

int y = 6;

int \*p;

p = &x;

p = &y;

On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.

int x = 5;

int y = 6;

int &r = x;

1. **Memory Address:** A pointer has its own memory address and size on the stack whereas a reference shares the same memory address (with the original variable) but also takes up some space on the stack. References may be passed to functions, stored in classes, etc. in a manner very similar to pointers.  Pointer is an independent variable and can be assigned NEW address values; whereas a reference, once assigned, can never refer to any new object until the variable goes out of scope.
2. **NULL value:**Pointer can be assigned NULL directly, whereas reference cannot. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into exception situation.
3. **Indirection:** You can have pointers to pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.
4. **Arithmetic operations:** Various arithmetic operations can be performed on pointers whereas there is no such thing called Reference Arithmetic.(but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in &obj + 5).)

**When to use What**

The performances are exactly the same, as references are implemented internally as pointers. But still you can keep some points in your mind to decide when to use what :

* Use references
  + In function parameters and return types.
* Use pointers:
  + To implement data structures like linked list, tree, etc and their algorithms.
  + Use pointers if  pointer arithmetic or passing NULL-pointer is needed. For example for arrays (Note that array access is implemented using pointer arithmetic).

Use references when you can, and pointers when you have to. References are usually preferred over pointers whenever you don’t need “reseating”. This usually means that references are most useful in a class’s public interface. References typically appear on the skin of an object, and pointers on the inside.

The exception to the above is where a function’s parameter or return value needs a “sentinel” reference — a reference that does not refer to an object. This is usually best done by returning/taking a pointer, and giving the NULL pointer this special significance (references must always alias objects, not a dereferenced null pointer).