CSE 333 Notes

Contents

1 Memory & Data

1.1 Pointers

Pointers are special variables that store addresses

- Since the length of an address is the word size, the size of a pointer is also the word size
- The address-of operator (`&`) returns the address of a variable in memory
- The dereference operator ($*$) accesses the data at the location of the pointer
- `type*` denotes the address of a variable of type `type`

1.2 Pointer Arithmetic

Pointer arithmetic takes the size of the data type being pointed to into account by automatically scaling the arithmetic operation

- i.e. Given `int* p`, the operation `p += 1` increases the value of `p` by 4, the size of `int`
- i.e. Given `long* p`, the operation `p += 1` increases the value of `p` by 8, the size of `long`
- i.e. Given `int* p1` and `int* p2`, the operation `p2 - p1` returns the number of `int` values that fit between the two addresses

1.3 Arrays

An array is a set of contiguous locations in memory that store the same type of data object

- Declaring an array follows the syntax `type array_name[num];` and sets aside `num * sizeof(type)` bytes of consecutive memory
  - `num` is the number of elements in the array

1.4 Array Subscript Notation

`array_name[n]` refers to the n^{th} element of the array, and is actually accessed via `*(array_name + n)`

- In the C programming language, `n` can be smaller than 0 or larger than `num`
  - This will access an address beyond the scope of the array

1.5 Strings

A string is represented by an array of characters that are terminated by the null character

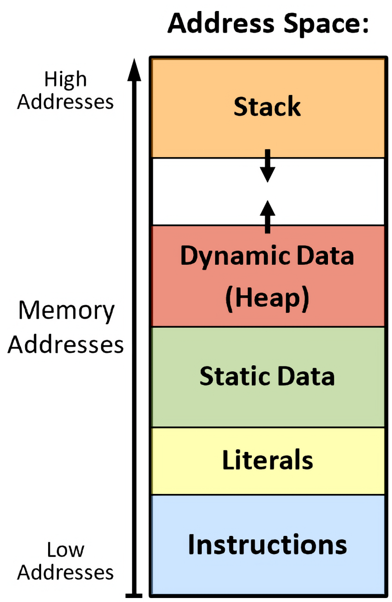
- i.e. `char str[] = "hello"` will allocate a character array of size 6 with the following content

'h'	'e'	'l'	'l'	'o'	'\0'

1.6 String Literals

A string literal is a sequence of characters surrounded by double quotes (i.e. `"hello world"`) and automatically stored in memory as an array of characters, terminated by '\0'

1.7 Memory Layout



Data:	Management:	Permissions:
Local variables and procedure context	Managed "automatically" (by compiler/assembly)	Writable Not executable
Variables allocated with `new` or `malloc`	Managed "dynamically" (by programmer)	Writable Not executable
Static variables (including globals)	Managed "statically" (initialized at process start)	Writable Not executable
Large literals/constants (*e.g.* "example")	Managed "statically" (initialized at process start)	Read-only Not executable
Program code	Managed "statically" (initialized at process start)	Read-only Executable

1.8 Significant Bits

- The left-most bit in a chunk of memory is the most-significant bit
- The right-most bit in a chunk of memory is the least-significant bit

1.9 Endianness

- In a big-endian machine, the most significant byte of data is stored at the lowest address
- In a little-endian machine, the least significant byte of data is stored at the lowest address

2 Programming in C

2.1 Generic Program Layout

```
#include <system_files>
#include "local_files"

#define macro_name macro_expr

/* declare functions */
/* declare external variables & structs */

int main(int argc, char* argv[]) {
    /* statements */
}

/* define other functions */
```

- `argc` is the number of command-line arguments passed
- `argv` is an array containing *pointers* to the command-line arguments which are represented as strings
- The executable name is passed in as an argument

2.2 Functions

```
/* declare function */
returnType fname(type param1, ..., type paramN);

/* call function */
int main(int argc, char* argv[]) {
    fname(arg1, ..., argN);
}

/* define function */
returnType fname(type param1, ..., type paramN) {
    /* statements */
}
```

- Functions must be declared before being called
  - Functions are often declared in header files and incorporated via `#include`
- Functions can only be defined once
  - Functions cannot be overloaded

2.3 Error Handling

- C does not have error handling, i.e. `try...catch` block
- Functions return errors as integer error codes
  - `CONSTANT_NAMES` are used to abstract away the integer error codes
- The global variable `errno` holds the value of the last system error

2.4 Status Codes and Signals

- Processes exit with status codes, e.g. `EXIT_SUCCESS` and `EXIT_FAILURE`
  - Standard codes can be found in `stdlib.h`
- Crashes trigger signals from the OS

2.5 Modules

A module is a self-contained piece of an overall program

- Has externally visible functions that can be invoked
- Has externally visible `typedefs` and global variables that can be used
- May have internal functions, `typedefs`, and global variables that are private
- Has an interface which declares the module's public functions, `typedefs`, and global variables

2.6 Header Files

A header file is a file which declares the interface to a module

- The header `main.h` corresponds to the source file `main.c`
- Holds the variables, types, and function prototype declarations that make up the interface to a module
- Enables the use of `#include` to import the module
- Documentation must be written in the header file

2.7 Header Guards

Header guards safeguard against redefining macros

```
#ifndef PAIR_H_

struct Pair {
    int a;
    int b;
}
#endif
```

- Use header guards to define macros everywhere they are used

2.8 Preprocessors

The C preprocessor is a sequential and stateful search-and-replace text-processor that transforms your source code before the compiler runs

- The preprocessor takes in a C file and outputs a C file
- The preprocessor processes the directives found in the file, i.e. `#include`, `#define`, `//comments`

2.9 Linkages

Linkages specify whether an identifier declared in different scopes or in the same scope more than once refer to the same object or function

- External linkage
  - `extern` makes a declaration externally visible
  - i.e. `extern int i;`
  - Global variables and functions are `extern` by default
- Internal linkage
  - `static` restricts a definition to visibility within the file
  - i..e `static int i = 0;`
  - It is good practice to use `static` when declaring local globals

2.10 Format Specifiers

Specifier	Output
`%d`	`Signed int`
`%u`	`Unsigned int`
`%o`	`Unsigned octal`
`%x`	`Unsigned hex int (lower)`
`%X`	`Unsigned hex int (upper)`
`%f`	`Decimal floating point`
`%a`	`Hexadecimal floating point (lower)`
`%A`	`Hexadecimal floating point (upper)`
`%c`	`Character`
`%s`	`String`
`%p`	`Pointer`
`%n`	`Return index of pointer`

- Use the `0N` modifier to print an int with at least `N` digits
  - i.e. `printf("%03d", 3)` outputs `003`
- Use the `0.N` modifier to print a float with exactly `N` decimal places
  - i.e. `printf("%03f", 3.14)` outputs `3.140`
- Use the `0N.M` modifier to print a float with at least $N - 1$ digits and exactly N decimal places
  - i.e. `printf("%07.4f", 3.14)` outputs `03.1400`

3 Memory Allocation

3.1 Null

`NULL` is a memory location that is guaranteed to be invalid

- Dereferencing `NULL` will cause a segmentation fault
- `NULL` is often used as an indicator of an uninitialized pointer or an allocation error

3.2 Dynamic Memory Allocation

`void* malloc(size_t size)` will request a continuous block of at least `size` bytes of uninitialized memory from the allocator

- `malloc` returns a pointer to the beginning of the allocated space, or `NULL` in case of an allocation failure
- Often `size` is calculated by use of the `sizeof()` macro
  - i.e. `int* ptr = (int*) malloc(n * sizeof(int))` allocates the space for an array of `n` `int`'s
- `calloc` is an allocation function that initializes the allocated heap block to all zeros
- `realloc` changes the size (up or down) of a previously allocated block

3.3 Dynamic Memory Deallocation

`void free(void* p)` will deallocate the entirety of the heap block pointed to by `p`

- `p` must hold the same address that was returned by the allocating function, `p` cannot point to an arbitrary byte within the heap block
- If `p` is `NULL`, then nothing happens
- Invalid calls to `free` will result in a system exception
  - i.e. address of the middle of a block, address of a previously deallocated block, address outside of the heap

3.4 Structs

A struct is a user-defined, structured group of variables. A struct definition is formatted as

```
struct struct_tag {
    type_1 field_1;
    ...
    type_N field_N;
};
struct struct_tag var_name;
```

- The user-chosen struct tag is part of the name of the new data type we are defining, which is the two-word name `struct struct_tag`
  - The struct tag can be omitted if the struct is not used elsewhere in the code
- Fields can be accessed from an instance using the '`.`' operator, i.e. `strct.x`
- Fields can be accessed from a pointer
  - By dereferencing and using the '`.`' operator, i.e. `(*ptr).x`
  - By using the `->` operator, i.e. `ptr->x`

3.5 Typedef

Typedef allows us to create aliases to other data types, such as two-word struct data type names

- Typedef statements are of the form `typedef <data type> <alias>;`

i.e. for structs, a typedef statement can be used after or combined with the struct definition

```
struct point_st {
    int x;
    int y;
};
typedef struct point_st Point;
Point pt1;
```

```
typedef struct point_st {
    int x;
    int y;
} Point;
Point pt1;
```

3.6 Use of Structs

- Structs can be copied by assigning an instance of a struct to another instance
- If the struct is small and we are only reading it, then passing a copy of the struct is faster. Otherwise use pointers

4 File I/O

4.1 Streams

A stream is a sequence of characters that flows to and from a device

- Streams can either be text or binary
- Streams are buffered by default
- Streams are manipulated with a `FILE*` pointer

4.2 Stream Functions

- `FILE* fopen(filename, mode);`
  Opens a stream to the specified file in the specified access mode
- `int fclose(stream);`
  Closes the specified stream and file
- `int fprintf(stream, format, ...);`
  Writes a formatted C string, similar to `printf(...)`
- `int fscanf(stream, format, ...);`
  Reads data and stores data matching the format string
- `size_t fwrite(ptr, size, count, stream);`
  Writes an array of `count` elements of `size` bytes from `ptr` to `stream`
- `size_t fread(ptr, size, count, stream);`
  Reads an array of `count` elements of `size` bytes from `stream` to `ptr`

4.3 Stream Error Checking & Handling

- `int ferror(stream);`
  Checks if the error indicator associated with the specified stream is set
- `int clearerr(stream);`
  Reset error and EOF indicators for the specified stream
- `void perror(message);`
  Prints `message` followed by an error message related to `errno` to `stderr`

4.4 Stream Buffering

By default, data written by `fwrite()` is copied into a buffer inside the process' address space

- At some point, the buffer will be drained into the destination
  - When the buffer size is exceeded
  - When `fflush()` is explicitly called
  - When `fclose()` is called
  - When the process exits gracefully
- Advantages of stream buffering
  - Buffering improves performance by reducing disk accesses
  - Buffering improves abstraction and convenience
- Disadvantages of stream buffering
  - Buffering reduces reliability since the buffer needs to be flushed
  - Buffering reduces performance by requiring extra copies from the buffer to the file

4.5 Portable Operating System Interface (POSIX)

POSIX is a set of lower-level file access APIs

- `int open(filename, mode);`
  Opens a stream to the specified file in the specified access mode
- `int close(stream);`
  Closes the specified stream and file
- `ssize_t read(int fd, void* buf, size_t count);`
  Advances forward in the file by number of bytes read and returns the number of bytes read

4.6 Directories

A directory is a special file that stores the names and locations of the related files/directories

- This includes itself (.), its parent directory (..), and all of its children
- The directory is accessible via POSIX

5 Programming in C++

5.1 Pointers

Pointers are special variables that store addresses

- Pointers are initialized via `int* x_ptr = &x;`
- Modifying the pointer does not modify what it points to
- Pointers in C++ work the same as in C

5.2 References

References are aliases for other variables

- References are initialized via `int& x_ref = &x;`
- Modifying the reference modifies the aliased variable
- `const` references are used for complex structs/object instances to reduce memory use

5.3 `const`

The `const` keyword indicates to the compiler that the associated variable cannot be mutated

- `const` next to the pointer name indicates that the pointer value cannot be mutated
  - i.e. `int* const ptr`
- `const` next to the data type indicates that the object being referenced by the pointer cannot be mutated
  - i.e. `const int* ptr`

5.4 Classes

- Objects can be declared as `const`
  - Once a `const` object has been constructed, its member variables cannot be changed
  - A `const` object can only invoke member functions that are labeled `const`
  - i.e. `const Name identifier`
- Object member functions can be declared as `const`
  - A `const` member function cannot modify the object it was called on
  - Member functions that do not modify the object should be marked `const`
  - i.e. `retType MethodName() const { // body statements; }`

5.5 Class Definitions

```
class Name {
 public:
  // public member definitions and declarations

 private:
  // private member definitions and declarations
};

// friend function definitions that use the class
```

- Class definitions are contained in a `.h` file
- Members can be functions or variables
- In-line setter and getter methods are also declared in the header file

5.6 Class Declarations

```
retType Name::MethodName(type1 param1, ..., typeN paramN) {
  // body statements
}
```

- Class declarations are contained in a `.cc` file

5.7 Constructors

A constructor (ctor) initializes a newly-instantiated object

- A class can have multiple constructors that differ in parameters
- A constructor must be invoked when creating a new instance of an object
- Classes often have a default constructor, an explicit constructor, and a copy constructor
  - The default copy constructor performs a shallow copy of all the fields

5.8 Destructors

A destructor (dtor) frees any dynamic storage or other resources owned by the object

- The destructor is automatically invoked when a class instance is deleted or goes out of scope

5.9 Rule of Three

If a destructor, copy constructor, or assignment operator is defined, then all three must be defined

5.10 Access Modifiers

Access modifiers control the visibility of fields, methods, and constructors in a class

- `public` members are accessible to all parts of the program
- `private` members are accessible to other member functions of the class
- `protected` members are accessible to member functions of the class and any sub-classes

`struct` members default to `public` and `class` members default to `private`

5.11 Non-Member Functions

Non-member functions are regular functions that make use of some class

- Non-member functions are called as regular functions instead of as a member of a class object instance
- Non-member functions do not have access to the class' private members
- Non-member functions are declared in the class' header file, but outside of the class definition

5.12 `friend` Non-Member Functions

A class can give a non-member function access to its non-`public` members by declaring it as a `friend` within its definition

- `friend` non-member functions have the same access privileges as a member function
- `friend` functions are usually unnecessary if the class includes the appropriate public getter functions

5.13 Namespaces

A namespace defines a scope to the identifiers inside it

- If a namespace already exists, then re-defining the namespace adds to the existing namespace

5.14 `new`/`delete`

To allocate memory on the heap in C++, use the `new` keyword instead of `malloc()`. To deallocate memory on the heap in C++, use the `delete` keyword instead of `free()`

- `new` never returns a null pointer
- i.e. `int* type_ptr = new int;` allocates a new primitive integer type and `delete type_ptr;` deallocates the type
- i.e. `Point* object_ptr = new Point();` allocates a new `Point` object and `delete object_ptr;` deallocates the object
- i.e. `int* arr_ptr = new int[size];` allocates a new primitive integer array and `delete[] arr_ptr;`

5.15 Templates

Templates enable functions or classes to accept generic types as parameters

```
template <typename T>
int compare(const T &value1, const T &value2) {
    ...
}
```

- Templated functions should be declared in the header file alongside their definitions

6 C++ Standard Template Library

6.1 Type Inference

Types can be inferred using the `auto` keyword

```
type Function();

int main(int argc, char* argv[]) {
    auto var = Function();
}
```

- The expression using `auto` must contain explicit initialization with defined return types

6.2 Containers

A container is an object that stores a collection of other objects in memory

- Containers store by value instead of by reference
- When an object is inserted, the container makes a copy
- Sequence containers index their elements numerically
  - i.e. `vector`, `deque`, `list`
- Associative containers index their elements by key
  - i.e. `set`, `map`, `multiset`

6.3 Lists

A list is a generic doubly-linked list

- Does not support random access
- Inserting and deleting elements is $O(1)$ time

6.4 Vectors

A vector is a generic dynamically resizable contiguous array

- Random access is $O(1)$ time
- Inserting and deleting elements from the end is amortized $O(1)$ time
- Inserting and deleting elements from the start or middle is $O(n)$ time

6.5 Maps

A map is a key/value table implemented as a search tree

- Elements are type `pair<key_type, value_type>` and are stored in sorted order
- `key_type` must support less-than operator
- Lookup and insertion is $O(\log n)$ time

6.6 Iterators

Each container class has an associated iterator class used to iterate through elements of the container

- Iterators range from `begin()` to `end()`, where `end()` is one past the last container element

7 Smart Pointers

7.1 Smart Pointers

A smart pointer is an object that stores a pointer to a heap-allocated object

- Smart pointers look and behave like a regular pointer
- Smart pointers will destroy and delete the associated object at the right time

7.2 `shared_ptr`

`shared_ptr` is a smart pointer that tracks the number of references to a piece of data, and only deallocates when no smart pointers are managing that data

- Constructors create the counter
- Copy constructors and assignment operators increment the counter
- Destructors decrement the counter

7.3 `unique_ptr`

`unique_ptr` is a smart pointer that is the sole owner of a pointer to a piece of data

- `unique_ptr` can release ownership of a pointer via `release()`
  - `x.release()` returns the pointer stored in `x` and replaces it with `nullptr`
- A new `unique_ptr` can accept ownership of a pointer via initialization
  - `y(x.release())` constructs a new `unique_ptr` with the pointer stored in `x`
- An existing `unique_ptr` can accept ownership of a pointer via `reset()`
  - `y.reset(x.release())` deletes the previous pointer and its associated data, and replaces it with the pointer stored in `x`
- `unique_ptr` can transfer ownership of a pointer via `move()`
  - `y = move(x)` interface is equivalent to `y.reset(x.release())`

7.4 `weak_ptr`

`weak_ptr` is similar to `shared_ptr` but does not affect the reference count

- `weak_ptr` cannot be directly dereferenced

8 Inheritance

8.1 Inheritance

A derived/child class inherits all the methods and properties from a base/parent class

```
#include "BaseClass.h"

class DerivedClass : public BaseClass {
    ...
}
```

- Constructors, destructors, copy constructor, and assignment operator are never inherited

8.2 Virtual Functions

A virtual function is a member function of a base class which may be redefined by derived classes

- It is best practice for derived functions to declare member functions as `virtual` where appropriate

8.3 Overriding Functions

Virtual functions can be overridden using the `override` keyword

- Declares to the compiler that this method should be overriding an inherited virtual function
- Prevents overloading vs. overriding bugs
- A member function cannot be declared as both `virtual` and `override`

8.4 Pure Virtual Functions

A pure virtual function is a member function of a base class which is only implemented in derived classes

- A pure virtual function is defined via `virtual retType MethodName() = 0;`

8.5 Abstract Classes

A class containing any pure virtual function is abstract

- Abstract classes cannot be instantiated
- Abstract classes must be extended and overridden in order to be used
- An interface is an abstract class containing only pure virtual functions

9 Casts

9.1 Static Casts

`static_cast<to_type>(expression)` performs well-defined conversions

- Can convert pointers between classes of related type
  - i.e. casting between `void*` and `T*`
- Can convert related references
  - i.e. casting between `float` and `int`
- Static casts are checked at compile time

9.2 Dynamic Casts

`dynamic_cast<to_type>` performs related conversions

- Can convert pointers between classes of related type
  - i.e. casting between `BaseClass*` and `DerivedClass*`
- Can convert references between classes of related type
  - i.e. casting between `BaseClass` and `DerivedClass`
- Dynamic casts are checked at both compile time and run time

9.3 Constant Casts

`const_cast<to_type>` adds or removes `const`-ness

- Can convert between `const` and non-`const`

9.4 Reinterpret Casts

`reinterpret_cast<to_type>` performs conversions between incompatible types

- Involves the low-level reinterpretation of the bit pattern
  - i.e. storing a pointer in an `int`