San Diego State University

Department of Computer Science

Instructor Notes

In
Python, Go, C, and C++

Prepared by Shawn Healey

CS 320
Programming Languages

Contents

Part I

Compilers and Interpreters

Chapter 1

Computing Environment

1.1 Terminal Window

The *terminal window*, or *console*, provides a non-graphical interface users may use to issue commands to the operating system. Prior to Xerox's introduction of the desktop environment, computer users interacted exclusively with the computer through simple text input. Movies from the eighties, like *War Games*, illustrate this interface. With the popularity of the application *Windows 3.1*, however, graphical interfaces quickly dominated the market. With *Windows 95* Microsoft fully integrated the operating system and graphical interface, so they became inseparable.

That said, the need for command-line input remains. When connecting to a remote web server, for example, transmitting a graphical interface requires additional bandwidth. For most management tasks, the administrator simply needs to perform a few basic functions. Why include mouse support if the only reason for logging in is to restart an application? Instead, a simple text interface allows developers to make changes and configure the server without the extra clutter icons and backgrounds introduce.

Getting to this interface varies based upon the machine you use, but almost every one provides one.

Linux and MacOS Simply searching for the *terminal* application reveals it to the user. It arrives pre-installed with the operating system. Even the Raspberry Pi offers its users a terminal window in this fashion.

Windows Users Microsoft strongly supports Linux. Microsoft employs human beings who develop software, and these humans love Linux. This appreciation for the operating system does not fade when they begin receiving a Microsoft paycheck. For many development tasks, its operating system interface works magic. Consequently, Microsoft integrated Linux support. Simply looking for Linux in the Microsoft Store produces both an Ubuntu variety and a Kali edition.

Tethered Appliances Proprietary devices, like the *Chromebook* and *Surface*, provide varying support for the terminal window. They frequently monitor activity and, in the stated interest of user security, limit functionality. For software development, these platforms rarely succeed. Although they are substantially more affordable than a general purpose computer, and they support consumer-level applications easily, students developing software on these devices typically encounter insurmountable technical hurdles.

Students on these machines would be well served migrating to a Raspberry Pi for software development. These devices support Python and they allow users a level of freedom none of the tethered devices support. It might appear that the Chromebook supports a compiler, as the semester advances and students begin to include additional libraries, the device's limitations appear.

1.2 Secure Shell

Connecting to the university server from a laptop or personally owned computer requires the student to use a communication tool. Similar to how early computer-hobbyists connected to one another through a dial-up modem, two computers need to establish a digital connection between each other. Today, we connect via the Internet or Intranet.

Linux and MacOS The `ssh` program ships with the base installation for these operating systems, so users need only open a terminal window (See Section 1.1 for details) and launch the program with the correct data.

```
ssh csscXXXX@edoras.sdsu.edu
```

Windows Windows 10 users running the subsystem for Linux may proceed from the Linux directions, for the subsystem *is* Linux. Other Windows users must use a terminal emulator to connect to the remote server. The program PuTTy provides sufficient terminal level access to the remote university server. This application does not provide a graphical user interface, nor does it permit rendering the account's desktop. The PuTTy terminal emulator simply supplies a text interface to the host computer.

Using this interface, one may create folders and files, issue commands, run applications, and perform any amount of work independent of a desktop environment. Files created through this interface appear on the host machine, so students need not transfer them later if they develop strictly through a text interface.

That said, the remote machine lacks access to the contents of *your* machine, so any source code there is inaccessible to the remote computer[1].

1.3 Secure Copy

How does one transfer work from one computer to another? Certainly, an eight-inch floppy disk, with its massive 1.2 *megabyte* capacity, allows software developers to quickly and easily share data between machines, but what if you don't live in the 1980s? What then? Physically transferring files between computers, like one would do with a floppy or USB disk, requires the machines or media to be within physical proximity of one another. It requires something *physical* to actually move between them. The logistics of the transfer frequently make this method impractical[2]

Instead of transmitting the ones and zeros to a USB drive, what if we sent them across the network to the intended target? Obviously, we must encrypt the traffic to avoid people from watching everything we do, but this solution allows us to push and pull data to any point on the planet without leaving our seat. We can copy our home work to the office computer without changing our clothing. For this, we use secure copy.

Linux and MacOS Most basic installations include `scp` by default. This serves as a testament to its importance. The basic command syntax appears in Figure 1.1.

Windows Still not using the subsystem for Linux? Alright. The easiest method for file transfer is through a GUI program. WinSCP and Cyberduck

[1]Unless something terrible/wonderful is happening on your home network.

[2]For a glimpse at terror, investigate the AWS Snowmobile. Per Amazon: You can transfer up to 100PB per Snowmobile, a 45-foot long ruggedized shipping container, pulled by a semi-trailer truck.

```
scp -r ./program1 csscXXXX@edoras.sdsu.edu:~/assign/lab1/
```

Figure 1.1: An example of a recursive directory copy of a local folder to the university server through SCP.

should provide students with the capability of transferring files from their personal computer to the Edoras server. FileZilla at one point provided a reliable tool, but they corrupted the installer to also bundle offers and ads with the application, so I recommend caution when interacting with this company.

1.4 Basic Unix Commands

Once in the terminal, one may issue properly formatted Unix-style commands. Linux and Unix are case-sensitive, so the command `mkdir` will not run if the user types `MKDIR` or `MkDir`. The same applies to the names of files.

List Files Students may display the list of files and directories in the current directory with the command: `ls`

The command supports additional command-line arguments which modify its behavior. By default, the command displays an abbreviated list of the directory's contents. This list omits *hidden* files. On a Unix system, one specifies a hidden file by beginning the file's name with a period. Thus, the file `read.me` displays, but the file `.config` does not. We benefit from this because it helps reduce the clutter we see after issuing the command.

Should one wish to see these other files, one need only include the proper optional flag: `ls -a`

The `-` designates the beginning of a list of options, and the '`a`' specifies that the command should display *all* the folder's contents.

A few special symbols designate important directory locations. This notation serves as shorthand for a file's full path. Typing `pwd` at the command prompt produces the *full* or *absolute* path. This is the complete and precise location uniquely identifying the location for a file or directory. Sometimes we use this format, but the *relative* path to a file offers an easier reference.

For example, a program may rely on several custom icons or a special directory for the input images. What happens when the user installs the program on a different machine in a different location? If the program expects images in `/home/shawn/images`, what happens when someone other

Command	Description
`ls`	List the current directory contents
`ls ../`	List the contents of the parent directory
`ls /tmp`	List the contents of the `/tmp` directory at the file-system's root
`ls ./tmp`	List the contents of the `tmp` directory contained inside the current directory
`ls -la`	List all the current directory's contents in long form

Figure 1.2: The use of the List command in Unix with some of its possible command-line arguments.

Command	Description
`~`	The current user's home directory
`../`	The directory *above* the current directory
`./`	The current directory

Figure 1.3: Common relative path uses.

than the user `shawn` executes the program? If the software developer used a relative path instead and looked for images in `./images`, then the program will search for them from the current directory.

Changing, Creating, and Removing Directories Frequently, we need to organize the files on our computer. Multiple programs use files with similar names, and we need a way to keep things in some semblance of order. To this end, the operating system supports the creation of *directories*. One may navigate through them through use of the `cd` command. Users may create directories through the `mkdir` command, and they may remove empty directories with the `rmdir` command.

Creating, Copying, and Moving Files One may create files through several methods. Word processing applications typically offer their users the option of creating a new file, but there exist easier methods from the command line. To create a simple file with nothing inside, Unix users may use the `touch` command. This creates a zero-byte file with the designated name.

Command	Description
`cp ./first.txt second.txt`	Creates a copy of the file `first.txt` and places it in the file `second.txt`
`cp -r ./assign ./working`	Creates a *recursive* copy of the directory `assign` and places it in the directory `working`
`cp -r ~/assign ./working`	Creates a *recursive* copy of the directory `assign`, located in the user's home directory, and places it in the current directory with the name `working`
`mv ./lab1.py ../lab1.py`	Moves the file `lab1.py` from the current directory to the parent directory
`mv ./assign1.py ./lab1.py`	Renames the file `assign1.py` to `lab1.py`

Figure 1.4: The use of the copy and move commands in Unix with some of its possible command-line arguments.

If, instead of starting from a totally empty file, you wanted to work on a copy of an original file, the `cp` command creates a duplicate with the designated name.

Chapter 2

Language Evaluation

What makes a good language?[1] Is Java better than C? Is Python better than MATLAB or Go? Should developers implement functional programming in their design, or does it simply represent the latest programming fad [2]. Each language offers its own unique set of features and syntactic sugar [3], and these language-specific features tend to support development of a certain style of program or to support programming in a particular fashion.

Many decades ago, Brad Cox and Tim Love built a decorator around the C language that included the extra features they desired. They worked to create a super-set of the original C language with additional support for classes. It retained C's feel while increasing its capability. Independently, Bjarne Stroustrup, limited by C's syntax, wanted object oriented support in a language, so he created a C-like language with support for classes.

These parallel efforts, both based upon C, produced Objective-C and C++[1] respectively. Each language offers optimization focused developers, like embedded and systems programmers, features useful within the domain. Both of these languages allow programmers to focus on micro-optimization, timing, and storage requirements, and unlike C, each offers support for classes.

2.1 Criteria

Readability When reading code, how quickly does it take for the underlying intent to become apparent? Is the language saddled with obscure notation or a large boilerplate one must learn before producing working

[1]Originally *C with Classes*, but Bjarne wisely abandoned the original cumbersome name.

Language	Features
COBOL	OS-agnostic, simple, business-focused, scalable
Python	Module system, dynamically typed, extensive third-party libraries
Go	Simple, statically typed, concurrency
Rust	Memory safety, high-performance, concurrency, easy-to-use
C	Fast, procedural
C++	Optimized code, class abstraction, high-performance
Lisp	Functional, simple, wide support
Scheme	Functional, lisp-like, Free
Elixir	Functional, clustered computers, internet applications
Java	Object-oriented, multi-threaded, garbage-collected

Figure 2.1: Characteristics of select programming languages. Each holds its own strengths and limitations.

```
int i = 0;
i++;
++i;
i = i + 1;
i += 1;
```

Figure 2.2: Feature multiplicity in the Java and C++ programming languages. How many ways does one need to increment a variable?

```
int val = (size() > 4) ? 3 : 20;
```

```
val = 3 if size() > 4 else 20
```

Figure 2.3: The Ternary operator syntax as implemented in Java, C, and C++ vs. Python.

software? Readability certainly influences how we create software, but it impacts a software product in its maintenance cycle as well, for fatal errors may lurk within obscure syntax.

Many design firms and open source projects require all software submitted for release meet a pre-defined specification [4] [5] [6] [7]. This document attempts to standardize how source code looks so that new developers quickly understand the mechanics behind the software and not the individual coding style selected by the programmer. A Java or C++ standard, for example, might specify that every conditional or iteration statement must use curly-brace notation even for single statements, or it could forbid recursion.

When a language offers multiple ways of accomplishing something, each developer tends to establish a personal-preference. In a rich language with many features, unfamiliar syntax may hide a task the reviewer or maintainer otherwise understands. Does the software engineer spend time fixing the active problem, or should the developer instead spend an afternoon searching incomplete threads on Stack Overflow from 2009?

Although many languages include the boolean data type, the abstraction remains absent from the C language which simply treats 0 as false. Consequently, C developers must define values for the terms 'true' and 'false' themselves, for doing so improves readability. Consequently, one may see `TRUE`, `true`, or `True` defined at various points in a C project based upon what files it included and the individual developer's preference.

```
list_comp = [n*n for n in range(10)]
gen_exprs = (n*n for n in range(10))
```

Figure 2.4: List comprehensions and generator expressions in Python offer different functionality, but the syntax difference appears subtle.

```
num = a * b
data = "hi" * 3

num = 1 + 1
data = "1" + "1"
```

Figure 2.5: These operations in Python perform different operations based upon the context of their data types.

As the number of keywords in a language increases, its readability decreases, for each keyword carries its own meaning, special cases, and proper uses. In mathematics, orthogonality captures the geometric concept of perpendicularity. When applied to computer science, the more orthogonal the language, the fewer exceptions the language rules require. The less redundancies in a language – the fewer ways there are to accomplish one task – the more orthogonal.

Writibility Languages pleasing to read tend to also provide enjoyable writing experiences. Thus, readability and writibility share many essential characteristics, but they remain distinct. A developer may find it easy to read software written by another developer in a language, yet the same individual might struggle when writing code in the language. Conversely, the extra *expressiveness* provided by offering many ways to accomplish the same task speeds up software development for some users.

Languages with redundant characteristics impairs readability, as mentioned previously, for the reader must obviously understand the syntax and behavior when interpreting the code. When writing code, on the other hand, these readability impairments act as writibility enhancements, for one quickly finds `x = x+1` annoying when `x++` does the trick.

Reliability Run-time errors create a poor user experience, risk security, decrease confidence in the product, and may represent a danger to human

```
with open("input.txt") as the_file:
    for line in the_file:
        print(line)
```

Figure 2.6: Opening a file in Python using the `with` keyword performs substantial work behind the scenes.

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)

func main(){
    dat, err := ioutil.ReadFile("input.txt")
    fmt.Print(string(dat))
}
```

Figure 2.7: Opening a file in Go.

```go
package main

import (
  "bufio"
  "fmt"
  "os"
  "strings"
)

func main() {

  reader := bufio.NewReader(os.Stdin)
  fmt.Print("Enter Text: ")
  text, _ := reader.ReadString('\n')

  scanner := bufio.NewScanner(os.Stdin)
  scanner.Scan()
  fmt.Println(scanner.Text())

}
```

Figure 2.8: Reading in strings from the standard input in Go.

```python
res = input("Enter an expression (using input):")
print(res)

res = raw_input("Enter an expression (raw_input)")
print(res)
```

Figure 2.9: The `input` method may not perform as the developer intends based on the Python version.

```
try:
   count = int(raw_input("Please enter an integer: "))
except ValueError:
   print("Error, wrong type.")
else:
   print("{} is an excellent number".format(count)
```

Figure 2.10: Exception handling in Python.

life. Detecting errors at compile time allows the developer to implement a recovery strategy directly in software to help prevent a catastrophic failure. Consequently, some languages, like Java, Python, and C++, include support for *exception handling*, for it allows programmers to specify how one should recover from a particular set of errors.

This additional functionality, however, requires processing support. Throwing an exception in Java, for example, results in substantial work, for the computer prepares a complete stack frame with the exception. Additionally, as more features migrate into the language its readability suffers. Java programmers must understand the difference between checked and unchecked exceptions, and the exception handling syntax introduces new keywords with different meanings.

Language features and hooks offer incentives that might make a language appear more reliable, but when these features make reading the language difficult, they open up the code to logical errors. Conversely, when one understands and easily reads a language, logic errors and code ambiguities stand out.

☞ The easier it is to read and write a language, the fewer errors and ambiguities appear in the code.

Type Checking Variable *type checking* also helps detect errors during compilation, but it introduces a significant set of keywords into the language. The added notation does, however, help developers and automated tools verify correct code. Alternatively, one might perform run-time type checking, but doing so impairs writibility, for people quickly tire when forced to constantly cast and test variables before using them.

Languages support different levels of type checking based upon the types of errors they attempt to detect and prevent. A strongly *statically-typed*

```
def printProps(label, thing):
   print("Value of {}: {}".format(label, thing))
   print("Type of {}: {}".format(label, type(thing)))


a = 1
printProps("a",a)
a = "hello there"
printProps("a",a)
a = True
printProps("a",a)
a = 3.50
printProps("a",a)
a = [x*x for x in range(10)]
printProps("a",a)
a = (x*x for x in range(10))
printProps("a",a)
a = lambda x: (x%2 != 0)
printProps("a",a)
```

Figure 2.11: Dynamic typing in Python. The variable `a` changes from an integer to a function all within the same scope.

language, for example, the compiler binds variable names to their distinct types at compile time. Consequently, they cannot change their types at run-time. Once introduced as a character, for example, the variable remains a character; the compiler prevents the variable from dynamically switching types and becoming something different. This type system helps identify problems in production code, for it helps ensure that even rarely traversed code branches use variables as intended.

In a *manifestly typed* language, developers must declare a variable's type in the source file. Java and many C-like languages use manifest typing. Consequently, one cannot introduce variables into the current scope without first explicitly declaring them to the compiler or, in languages supporting *type-inference*, using them in a way in which it might infer the type.

Cost Any languages, be it C++ or Elixir, carries a cost. It may appear directly in the form of an integrated development environment, for professional tools carry costs, or indirectly through the additional time it takes to begin developing in the language. Some languages and modern machine learning

```
package main

import (
   "fmt"
)

var (
   IsReady  bool   = false
   MaxInt uint64     = 1<<64 - 1
   MainUser string   = "Assistant"
)

func main() {

    var a = "Aztec"

   fmt.Printf("Type: %T Value: %v\n", IsReady, IsReady)
   fmt.Printf("Type: %T Value: %v\n", MaxInt, MaxInt)
   fmt.Printf("Type: %T Value: %v\n", a, a)
}
```

Figure 2.12: Some of the basic types in Go.

```
package main

import "fmt"

func main() {
   var OneWay = [3]int{1, 2, 3}
   Another := [3]int{1,2,3}
   Variety := [...]string{"let","me","burn","you"}

   fmt.Println(OneWay)
   fmt.Println(Another)
   fmt.Println(Variety)
}
```

Figure 2.13: Go also supports the derived type array for sequential access.

tools carry strict hardware requirements, for one cannot useful calculations using CUDA [2] without a compliant Graphics Processing Unit (GPU). With internet based computing and software-as-a-service, costs might appear as CPU time.

Unhappy developers negatively impact an organization's morale, so forcing them to work in a language ill-suited to the task or with limited development support tools may increase employee turnover. This effectively erases any undocumented project history. Motives behind design decisions begin disappearing, and rediscovering the reason behind software can appear cryptic, for developers rarely document every aspect of a complicated system.

2.2 Compiled and Interpreted

Ahead-of-time compilation is the act of compiling a higher-level programming language such as C or C++, or an intermediate representation such as Java bytecode or .NET Framework Common Intermediate Language (CIL) code, into a native (system-dependent) machine code so that the resulting binary file can execute natively.

Just-in-time compilation is a way of executing computer code that involves compilation during execution of a program – at run time – rather than prior to execution. Most often, this consists of source code or more commonly bytecode translation to machine code, which is then executed directly.

A common implementation of JIT compilation is to first have AOT compilation to bytecode (virtual machine code), known as bytecode compilation, and then have JIT compilation to machine code (dynamic compilation), rather than interpretation of the bytecode. This improves the run-time performance compared to interpretation, at the cost of lag due to compilation. JIT compilers translate continuously, as with interpreters, but caching of compiled code minimizes lag on future execution of the same code during a given run. Since only part of the program is compiled, there is significantly less lag than if the entire program were compiled prior to execution.

An Interpreter is a computer program that directly executes instructions written in a programming or scripting language without requiring them previously to have been compiled into a machine language program. An interpreter generally uses one of the following strategies for program execution:

[2]Originally an acronym for Compute Unified Device Architecture (CUDA)

parse the source code and perform its behavior directly;

translate source code into some efficient intermediate representation and immediately execute this;

explicitly execute stored pre-compiled code made by a compiler which is part of the interpreter system.

While interpretation and compilation are the two main means by which programming languages are implemented, they are not mutually exclusive, as most interpreting systems also perform some translation work, just like compilers. The terms ”interpreted language” or ”compiled language” signify that the canonical implementation of that language is an interpreter or a compiler, respectively. A high level language is ideally an abstraction independent of particular implementations.

2.3 Typing

Strong and Weak Typing Languages fail to cleanly subdivide into strong and weak type systems, for there exists no agreed upon definition of the terms. Instead, the question of if a language is strongly or weakly typed asks *how* strongly typed a language appears. In a strongly typed language, once the software designates a particular region of memory as a specific type (e.g., `int`, `double`, `String`, etc . . .) it remains that way throughout its life. Given this definition, one might view Java as a strongly typed language. That said, Java performs implicit casting of variables, and this technically weakens its type system.

Static Typing Many programming languages (e.g., Java, C, and C++) verify that the type of a variable is appropriate during the compilation process. Statically typed languages prevent software from changing the type of a variable within a block/scope of code during execution. This allows these tools to catch type mismatches at compile-time rather than during program run-time.

Dynamic Typing Dynamic typed languages verify the type of a variable at run-time during program execution. Variables may change their type within the same scope in this type system.

Manifest Typing Some languages reduce their exposure to certain types of software errors by requiring developers to declare every variable prior to use. Consequently, in these type systems, subtle misspellings appear as compile-time errors.

Type Coercion In the course of development, it may become necessary to treat a variable of one type as if it were another. One might wish to treat a floating point number as an integer during a computation or transform a simple integer into a floating point number in another. Someone might wish to treat non-zero values for a variable as True and 0 as False. We call this process *casting* or *type coercion*. An *implicit* type cast may happen when one assigns the literal value of 3 to a variable expecting a floating point number. The computer may safely up-cast the integer to 3.0 without losing any precision. It is implicit because it requires no intervention from the developer and may not even produce a warning. When the transformation may produce a loss of precision, however, languages typically require a different style of cast. An *explicit* cast tells the machine to treat a variable of one type as if it were that of another, just as an implicit cast, but these casts may result in a precision loss.

2.4 Domain Types

- General-purpose language (GPL): broadly applicable across application domains, and lacks specialized features for a particular domain.
- Domain-specific language (DSL): A programming language specialized to a particular application.

Domain-specific languages can help to shift the development of business information systems from traditional software developers to the typically larger group of domain-experts who (despite having less technical expertise) have deeper knowledge of the domain. However, Non-technical domain experts can find it hard to write or modify DSL programs by themselves.

Specialized languages require education, which takes time, but they may have a limited applicability, so it may not be time well spent.

2.4.1 Unix Shell Scripts

Unix shell scripts give a good example of a domain-specific language for data organization. They can manipulate data in files or user input in many different ways. Domain abstractions and notations include streams (such as stdin and stdout) and operations on streams (such as redirection and pipe).

These abstractions combine to make a robust language to describe the flow and organization of data.

```python
#!/usr/bin/env python3

import subprocess
host = raw_input("Enter a host to ping: ")

# Set up the echo command and direct the output to a pipe
p1 = subprocess.Popen(['ping', '-c 2', host],
    stdout=subprocess.PIPE)

# Run the command
output = p1.communicate()[0]

print output
```

ref: https://www.pythonforbeginners.com/os/subprocess-for-system-administrators

2.4.2 Markup Languages

html and xml, and even LaTeX

wiki: A markup language is a system for annotating a document in a way that is syntactically distinguishable from the text. A common feature of many markup languages is that they intermix the text of a document with markup instructions in the same data stream or file. This is not necessary;

2.5 Exercises

1. The Java programming language includes substantial conflicting syntax. Provide an example of a symbol or keyword with multiple meanings in the language.

2. The Go language developers eliminated many features common in previous programming languages. Provide an example of at least one such feature. Do you agree code reads cleaner without the additional tool?

3. The example of dynamic typing in Figure 2.11 uses substantial repetition, for the call to `printProps` never changes. Rework this code such that it uses a loop and the call to `printProps` appears only once in the code.

Chapter 3

Computer Architecture

3.1 von Neumann Architecture

- Arithmetic/Logic Unit – holds the math co-processor. The circuitry which operates on the data. Performs the math and logic operations as the name implies.
- Control Unit – a component of a computer's central processing unit (CPU) that directs the operation of the processor. It tells the computer's memory, arithmetic and logic unit and input and output devices how to respond to the instructions that have been sent to the processor.
- Registers – quickly accessible location available to a computer's central processing unit (CPU). Registers usually consist of a small amount of fast storage, although some registers have specific hardware functions, and may be read-only or write-only

The von Neumann bottleneck Limited throughput because of the speed of accessing the memory unit. Reading from registers in the CPU is fast, but loading them with data from memory is relatively slow compared to the speed the processor is able of operating instructions.

> Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied

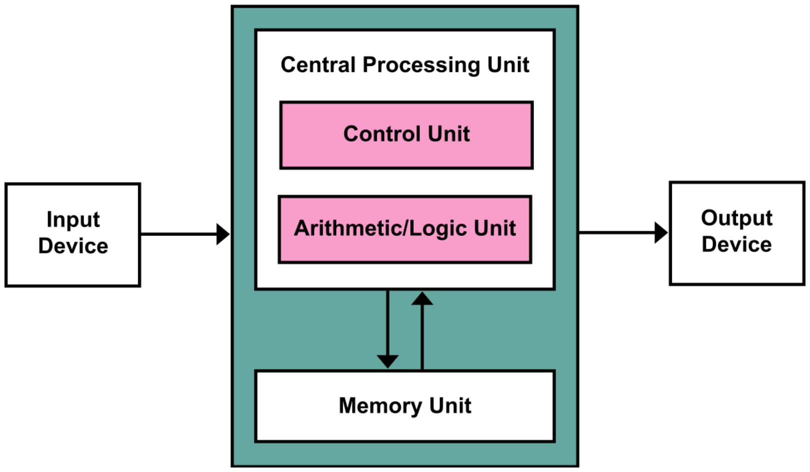


Figure 3.1: Example of von Neumann architecture[8]

.

to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it. – *John Backus in his 1977 ACM Turing Award lecture*[2]

Fetch-execute-cycle each instruction to execute must be fetched from memory by the cpu. the Program Counter maintains the address of the next instruction to execute.

```
initialize the program counter
repeat forever
  fetch the instruction pointed to by the pc
  increment the pc to point at the next instruction
  decode the instruction
  execute the instruction
end repeat
```

Imperative Languages modeled on machine architecture. Data and programs are in same memory and CPU executes instructions.

3.2 Implementation Methods

3.2.1 Machine Language

Machine Language uses instructions defined by vendor. Simple set of instructions to make the processor perform as desired. Low-level, but a gateway to higher languages.

Instruction set architecture serves as the interface between software and hardware. Defines everything a programmer, using machine language, would need to use the processor. Typically defines things like data types supported and the state data (i.e., main memory and registers). Different hardware implementations, (e.g., Intel vs. Atmel), can still use the same instruction set. The *interface* provides some vendor-neutrality by allowing other abstractions to swap out the hardware without requiring new, custom software.

- Complex Instruction Set Computer (CISC): is a computer in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions. The Motorola 68000 family of processors uses a CISC.
- Reduced Instruction Set Computer (RISC): is one whose instruction set architecture (ISA) allows it to have fewer cycles per instruction (CPI) than a complex instruction set computer (CISC). Various suggestions have been made regarding a precise definition of RISC, but the general concept is that such a computer has a small set of simple and general instructions, rather than a large set of complex and specialized instruction

An Opcode is the portion of a machine language instruction that specifies the operation to be performed. Beside the opcode itself, most instructions also specify the data they will process, in the form of operands. In addition to opcodes used in the instruction set architectures of various CPUs, which are hardware devices, they can also be used in abstract computing machines as part of their byte code specifications.

Bytecode is a form of instruction set designed for efficient execution by a software interpreter. Java bytecode or .NET Framework Common Intermediate Language (CIL) code

baload	33	arrayref, index -> value	loads a byte or Boolean value from an array
imul	68	value1, value2 -> result	multiply two integers
return	b1	-> [empty]	return void from method

Figure 3.2: Example Java bytecodes [9]

Java Byte Code is the language to which Java source is compiled and the Java Virtual Machine understands. Unlike compiled languages that have to be specifically compiled for each different type of computers, a Java program only needs to be converted to byte code once, after which it can run on any platform for which a Java Virtual Machine exists.

Bytecode is the compiled format for Java programs. Once a Java program has been converted to bytecode, it can be transferred across a network and executed by Java Virtual Machine (JVM). Bytecode files generally have a .class extension. It is not normally necessary for a Java programmer to know byte code, but it can be useful. [9]

Chapter 4

Context Free Grammars and Parse Trees

Context Free Grammars provide recursive rules for generating or validating strings in a language. Each rule, or *production*, simply replaces the string text. The language supports *terminal* symbols, which permit no further replacement or recursion, and *non-terminal* symbols. The CFG continues performing string replacements as long as productions activate. A CFG detects a valid string when it successfully resolves all non-terminal symbols into their eventual terminal replacements. Conversely, when a candidate string possesses non-terminal symbols, but the grammar cannot resolve any of the non-terminals with productions, it considers the string invalid.

Using this notation, one might define a grammar for detecting palindromes or numbers.

4.1 Backus-Naur Form

$\langle \textit{number} \rangle ::= \langle \textit{digit} \rangle | \langle \textit{digit} \rangle \langle \textit{number} \rangle$

$\langle \textit{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

4.2 Parse Trees

Given the grammar defined in Figure 4.2, we may begin generating valid strings. An `assign` always requires an `id` and an `expr`; the `assign` itself lacks direct recursion. Thus, we may construct the simplest compliant

$\langle number\rangle$ ::= $\langle digit\rangle$|$\langle digit\rangle\langle number\rangle$|-$\langle number\rangle$

$\langle digit\rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 4.1: An initial attempt to include negative numbers in the BNF leaves open some interesting strings.

$\langle assign\rangle$ ::= $\langle id\rangle$ = $\langle expr\rangle$

$\langle id\rangle$ ::= A | B | C

$\langle expr\rangle$ ::= $\langle id\rangle$ + $\langle expr\rangle$ | $\langle id\rangle$ - $\langle expr\rangle$ | $\langle id\rangle$ * $\langle expr\rangle$ | ($\langle expr\rangle$) | $\langle id\rangle$

Figure 4.2: An assignment statement's basic grammar.

$\langle expression\rangle$ ::= $\langle expression\rangle$ + $\langle term\rangle$ | $\langle expression\rangle$ - $\langle term\rangle$ | $\langle term\rangle$

$\langle term\rangle$ ::= $\langle term\rangle$ * $\langle factor\rangle$ | $\langle term\rangle$ / $\langle factor\rangle$ | $\langle factor\rangle$

$\langle factor\rangle$ ::= $\langle primary\rangle$ ^ $\langle factor\rangle$ | $\langle primary\rangle$

$\langle primary\rangle$ ::= $\langle primary\rangle$ | $\langle element\rangle$

$\langle element\rangle$ ::= ($\langle expression\rangle$) | $\langle number\rangle$

Figure 4.3: Backus-Naur Form for expressions in a simple computer language.

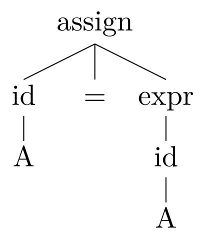


Figure 4.4: A parse tree for a simple assignment statement: `A = A`.

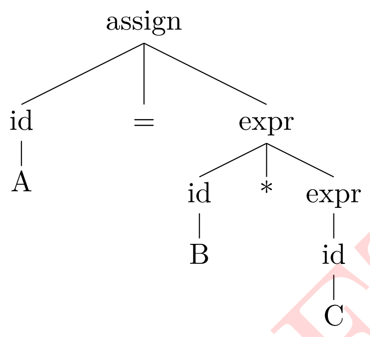


Figure 4.5: A parse tree for a simple assignment statement: `A = B * C`.

`assign` by using an `id` on the left and right side of the assignment. Thus, the string `A = A` satisfies the grammar. Visually, we may see this as shown in Figure 4.4

4.3 Evaluation Strategies

Eager Evaluation All of the programming languages taught at this point in the undergraduate curricula evaluate the values of the expressions when they are bound to a variable. When making a function call, prior to the machine performing the work defined in the function, the computer first evaluates the values it may use in the function. In computer science, we treat this behavior as *strict* or *eager* evaluation.

Lazy Evaluation Many functional programming languages, however, delay evaluating an expression until it is needed. That is, one might pass three parameters to a function without actually knowing their current values. Only when the function uses these values does the machine compute the expressions. This delays the calculation and has a side effect of eliminating

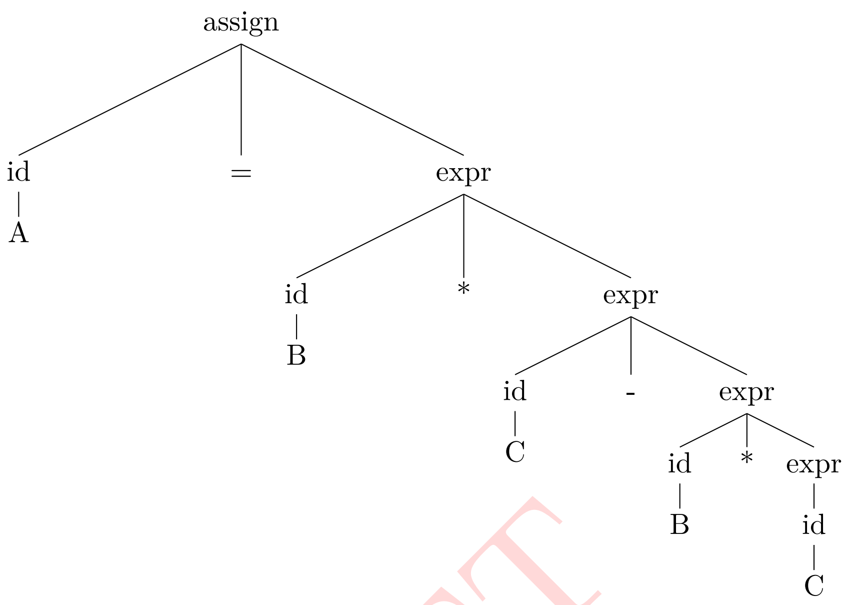


Figure 4.6: A parse tree for an assignment statement: `A = B * C - B * C`.

unnecessary computation. With lazy evaluation, one may define potentially infinite data structures, and the computer only calculates the set of values it uses.

Short-Circuit Evaluation Java developers familiar with the && operation may recall that the language explicitly begins evaluating if the operands are `true` or `false` from the left to the right. The first time it encounters a value that evaluates as `false`, the language knows the logical expression cannot ever become `true`, so it stops processing the operands and evaluates to `false`. Alternatively, the || operation stops processing operands the first time one of them evaluate as `true`.

Dynamic Programming Somewhat similar to lazy evaluation, Dynamic Programming defines a programming strategy where the software stores the results of previous computations in a map. Before doing the work defined in the function, it checks the map to see if ever performed the computation. If so, rather than doing the potentially cumbersome labor defined in the function, it returns the value it finds in the map. For new values, a function may have $O(n^3)$ performance, but old values simply require a map lookup which is $O(1)$.

4.4 Precedence and Associativity

Operator Precedence When evaluating an expression with multiple operators, in what order does one process them? From classical mathematics, we understand a basic order of operations where subtraction and addition happen after multiplication and division, but mathematical conventions do not always align with programming language realities. Consequently, most languages provide a table identifying operator precedence to make it clear where and if the language differs from convention.

Operator Associativity Multiple operators may appear in a single assignment, but what happens when the operators possess the same priority? Does 100/10/2 evaluate to 20 or 5? Along with the table of operator precedence, the language also establishes its operator associtivity. For each operator, the language specifies if it evaluates expressions from left-to-right or right-to-left.

4.5 Exercises

1. Create a program to identify the unique tokens in a file. The program shall prompt the user for a file name at program execution. The program shall read ASCII strings from the file until it reaches the EOF symbol. From the accumulated input, the application shall then extract only its unique tokens, separated by the space character, and write these to a new output file. For example, if provided with a file containing a text document, it shall write to the output file, with white-space between each token, every *unique* word contained therein. The order the words as they appear in the output does not need to be sorted in any particular order.

Program Name:	unique
Command Line Args:	none
Program Input:	Prompt user for input file name
Program Output:	Input file name with `out` as the extension
Allowed Imports:	Go: `bufio fmt io/ioutil os container/list`

2. Create a program to randomly combine four tokens found in an input file producing an output token with the four words squeezed together (e.g., somewhatlikethisbutrandom). It shall prompt the user for a file name at program execution. Using the name provided at run-time, the program shall read in the entire set of words to use during the combination process. After reaching the EOF symbol, the program shall then begin randomly selecting words and combining them into a concatenated output word. The program produces only one *crunched* output word for each execution. There shall be no duplicate words in any single output token (although combining 'racecar' with 'car' could make it appear so erronelusly). The program may ignore words in the input file with a length less than four.

Program Name:	crunch
Program Input:	`cin`
Program Output:	`cout`
Allowed Imports:	Go: `bufio fmt io/ioutil os container/list`

Example Output:

Valid
`reducefishersplendidProgram`
`windowsomlette,lethargicamount`

Invalid	
`hospitable chicken repair overflow`	Spaces between input words
`generatecarburetorRepairgenerate`	The token `generate` appears twice

3. Modify the program described in Problem 2 to accept a command line argument which specifies the number of strings the program outputs in a given execution. The program shall crunch together four words `n` times for the current execution.

4. Given the following simple grammar, indicate if the following sentences are valid with a `T` for *true* or `F` for *false*.

$$\langle S\rangle ::= \text{x}\langle A\rangle\langle B\rangle\text{y} \mid \text{z}\langle S\rangle\text{x}$$

$$\langle A\rangle ::= \text{y}\langle A\rangle\text{y} \mid \text{y}$$

$$\langle B\rangle ::= \text{z}\langle B\rangle \mid \text{zz}$$

   (a) zzzxyzzyxxx
   (b) zzxyyyzzyx
   (c) xyzy
   (d) xyzzy

5. Provide a parse tree for each valid sentence in Problem 4.

6. Produce a parse tree using right-to-left associtivity for the expression: 100/10/2

Chapter 5

Python

5.1 Integrated Development Environment

As a sign of Python's importance, many operating systems include a Python interpreter by default. Users may access it directly through the terminal window. Developing advanced programs entirely through the terminal window intimidates many new developers, and it lacks some additional tools or it hides them behind obscure command sequences one must memorize. Consequently, several vendors supply graphical implementations with support for more sophisticated development.

The *Anaconda* platform supplies a reliable Python development environment on all major operating systems. The free edition of the application includes packages essential to data science directly with the install. Other implementations require the user to download and configure the additional packages. Although the process remains easy, it requires a few additional steps we can simply avoid by using the Anaconda platform.

Alternatively, *PyCharm* by Jetbrains provides an interface similar to their IntelliJ Java development package. Maintained by a corporation, the development environment supplies many production-level features experienced developers enjoy.

Finally, the *Jupyter Notebook* presents its users with a web-style interface. This tool allows one to intermix paragraphs of text with small executable code sections. Although an excellent lecturer tool or presentation support device, it does not produce stand-alone python files. Although it serves an excellent proof-of-concept or discussion tool, it will not suffice for the programs we develop in this course. Feel free to use it to explore the language, but it will not produce a deliverable product.

5.2 Language Introduction

5.2.1 Comments

Lexical analyzer identifies and strips comments from source code. To indicate comments in python, one:

```
badUname = '" or ""="' # Malformed SQL
```

Block Comments The PEP-8 Style guide states that, ”Each line of a block comment starts with a # and a single space.” The Python language does not support multi-line block comments.

Inline Comments The PEP-8 Style guide advises developers, ”use inline comments sparingly.”

5.2.2 Variables and Data Types

Variables allow developers to abstract away the concept of a value. Rather than constantly referencing the specific address or register containing a value, one may assign it to a meaningful name. Not all languages support developer-specified variables. Very low-level languages, like assembly, allow programs to store data in pre-defined registers, but many do not support programmers and software developers arbitrarily assigning names to values.

Medium and high level languages allow programmers to specify a custom name for a memory space where one may store a value. Variables help improve code readability, for one need not constantly look back and try to determine what register one held vs. the contents of register two. Python supports variables and good coding practice encourages their appropriate use.

Valid Names Valid variable names in Python must meet a few strict requirements. First, one may not use the name of an existing keyword. Section 5.2.3 specifies each of the Python 3 keywords and briefly discusses their use. Simply put, however, when the Python interpreter encounters a keyword, it takes special action and verifies the word meets certain syntax requirements. Consequently, one may not use any of these as variable names. The machine lacks the ability to easily distinguish between `raise` as the name of a particular value and `raise` as a command to throw an exception.

```python
first = 1
second = 2
print(type(first))
print(type(second))
print("Changing types . . . ")
first = '1'
second = "2"
print(type(first))
print(type(second))
```

```
<type 'int'>
<type 'int'>
Changing types . . .
<type 'str'>
<type 'str'>
```

Figure 5.1: Python source(*left*) and its associated output (*right*). Python supports dynamic typing, so developers may change a variable's type in the same scope. Unlike Java, Python did not require the code to specify `first` as an integer.

In addition to avoiding existing keywords, a variable name may only include a specific set of characters. One may not, for example, use the '+' symbol in a variable's name, for when the Python interpreter encounters the '+' character, it assumes the user wishes to perform an addition operation. Every Python variable must begin with either an alphabetic character (either upper or lower case) or the underscore character.

Data Types Manifest-typed languages like C and Java require developers to explicitly specify a variable's *type* in source code. Static typed languages prevent variables from changing their type within the current scope. Python supports dynamic-typing, and it does not use manifest-typing.

Python includes built-in support for many of the traditionally *primitive* data types C-style programmers understand: boolean, integer, float, and string. Boolean values may hold either `True` or `False`, and developers frequently use these in conditional statements and as the return value for meaningfully named functions. Integers store whole numbers whereas floats hold floating point numbers. Because Python uses dynamic typing, however, changing between them is possible in the same block of code.

Additionally, Python supports more complicated structures like lists, maps, and sets without the need for additional includes or library files. The language itself builds in support for these essential data structures.

```
data = [True,64,8.9,"Delete",[1,2],(1,2)]
for item in data:
  print(type(item))
```

```
<class 'bool'>
<class 'int'>
<class 'float'>
<class 'str'>
<class 'list'>
<class 'tuple'>
```

Figure 5.2: Examples of Python's data types. This code (*left*) uses iteration in a `for` loop to print (*right*) the type of each item inside the `data` array.

and	del	global	not	while
as	elif	if	or	with
assert	else	import	pass	yield
break	except	in	print	
class	finally	is	raise	
continue	for	lambda	return	
def	from	nonlocal	try	

Figure 5.3: The keywords used by Python 3.

5.2.3 Keywords

Python 3 identifies slightly over thirty *keywords* as shown in Figure 5.3. Each of these words evokes unique behavior from the Python interpreter. That is, these words carry significance, and the Python interpreter must act in a specific way when it encounters these words. Every modern programming language includes keywords which help specify the lexicon and syntax in a language.

Keywords help define the basic actions one may perform with a language without including additional libraries. Keywords carry significance in code.

5.2.4 Operators and Operands

In a pinch, users may use Python as a basic calculator. Open the Python interpreter and enter any expression like $3 + 50$ and it will produce the correct result. Python supports basic math through a few special operators, but because programs work with a wide variety of data types, the meaning attached to these operations changes slightly. For example, dividing five by

Operand	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division: floating point division (Python 3)
`//`	Floor division: integer based division (Python 3)
`%`	Modulus: remainder after dividing the left side by the right
`**`	Exponentiation: Raise digit on the left side to power on the right

Figure 5.4: Arithmetic operators in Python. These operations produce a number after evaluation.

Operator	Description
`>`	Greater than
`<`	Less than
`==`	Strictly equal
`!=`	Not-equal
`<=`	Less than or equal
`>=`	Greater than or equal
`**`	Exponentiation: Raise digit on the left side to power on the right

Figure 5.5: Comparison operators in Python all resolve to either `true` or `false`.

two on a calculator produces 2.50, but the answer may, in fact, be simply 2 depending on what division operation the developer used.

This becomes more complicated when we apply these operations to strings. Dividing or performing the modulus on the string, ”Hello there” seems ambiguous or even meaningless. What would the expression ”OB” % 1 produce? Other operations, however, do make sense. Adding two strings together produces a new string with the second added to the end of the first. Programmers may even mix strings and numbers, for the Python interpreter knows to convert the number to a string representation before performing the operation.

- ”I want my ” + 2 + ”dollars” : NO – cannot mix int and string
- ”never gonna’ give ” * 2 : YES – repeat the string 2 times
- ”High” + ”ground” * 3 : YES – Highgroundgroundground

Operator	Description
`and`	Statement on left and statement on right must evaluate as `true`
`or`	Either or both statements must evaluate as `true`
`not`	Invert the boolean expression on the right

Figure 5.6: Logical operators in Python use only three keywords.

Operator	Description
`|`	If a one appears in either the left or right operand at that bit position, the output shall include a one in that position.
`&`	A bit position evaluates to a one only if both the left and right have a one in that position.
`~`	Invert the ones and zeros in the value to the right.
`^`	Exclusive OR. Either the left or right has a one in a given bit position, but not both.
`>>`	Bit-shift to the right (division) the number on the left the number of bit positions specified on the right.
`<<`	Bit-shift to the left (multiplication) the number on the left the number of bit positions specified on the right.

Figure 5.7: Bitwise operators in Python work at the binary level.

Operator	Use	Equivalent
`=`	`x = 2`	
`+=`	`x += 2`	`x = x + 2`
`-=`	`x -= 2`	`x = x - 2`
`*=`	`x *= 2`	`x = x * 2`
`/=`	`x /= 2`	`x = x / 2`
`//=`	`x //= 2`	`x = x // 2`
`**=`	`x **= 2`	`x = x ** 2`
`&=`	`x &= 2`	`x = x & 2`
`|=`	`x |= 2`	`x = x | 2`
`>>=`	`x >>= 2`	`x = x >>2`

Figure 5.8: Assignment operators in Python.

5.2.5 Functions

I've always objected to doing anything over again if I had already done it once. *–RDML Grace Murray Hopper*

Functions help simplify code by allowing developers to capture small, repeatable tasks or routines. Take the math functions, for example. The need to determine the sin or tangent of an angle is a common task. Since we frequently use them, it makes sense to break these off into their own functions.

Return Values Unlike the C-style languages, Python expects no return type in a function's signature. That said, `return` remains a keyword in the language, and it performs the same basic action. When a Python function encounters the return statement, it stops executing the function and places whatever value the return statement specifies on the call stack.

In Python, one function may return multiple different types. That is, it could return a floating point number in some situations and a string in others. Additionally, you may return *multiple* values from a single function call.

5.3 Recursion

```python
def azarath( m, z ):
  if m == 0:
    return 0
  elif m < 0:
    return -(z - azarath(z, m+1))
  return z + azarath(z, m-1)
```

Part II

Procedural and Object Oriented

Chapter 6

Introduction to C

6.1 Tools

Although numerous C compilers exist, (e.g., Turbo C and Borland C), this course uses freely distributed solutions that work at the command line level. For the type of programming done in this class, we need not use a complicated IDE.

6.1.1 Connecting to the Server

Students must access the Edoras server at San Diego State University via a terminal window. Linux and OSx users use the built-in shell, but Windows users must install additional software.

For terminal emulation, the application PuTTy meets our needs. Alternatively, students with the Windows Subsystem for Linux installed can install Ubuntu (and other Linux builds) into their Windows machines. These are not virtual machines.

server: `csscXXXX@edoras.sdsu.edu`

type: ssh

6.1.2 Compilers

Several compilers are available to students on a Linux system. They typically adhere to the same command line interface to attract new users already familiar with the compilation process.

- CC - Alias to the installed C Compiler (typically)

- GCC - GNU Compiler Collection [1]
- clang

Command Line Options

Flag	Result	Output
-g	Include debug information (GDB)	
-E	Simply run the pre-processor	Screen
-S	Pre-process and assemble	a.s
-o <file >	Specify output file	<file >
-P	Enable line-marker in -E	
-I <dir >	Add directory to include search path	
-c	Pre-process, compile, assemble	a.o

6.1.3 Debugger

The GNU suite includes a command line debugger to aid in simple programming or when one wishes to walk through assembly in a program.

Useful Commands

list	Display the current program
break <function name >	Halt execution after entering the specified function
b5	Break at line 5
run	Begin execution
s, step	Step INTO a function
n,next	Step OVER a function
print <var name >	Display value of associated variable

6.2 First Program

A trivial, introductory program typically prints something to the standard output that does not greet anyone or the planet.

```
#include <stdio.h>

int main(){
  printf("Party Cannon!");
```

[1]Identify the non-terminal in the name

```
  return 0;
}
```

This example, however, includes *substantially* more than the minimum C program requires, for it includes the standard input output libraries. To witness the impact of this decision on the code, compile the file with the `-E` option, for this causes GCC to stop compilation after pre-processing. In C, the pre-processor expands all macros and stitches in the external libraries and code referenced in the current module.

From the command prompt on a UNIX system, piping the output of the compilation process to the word count tool yields the number of output lines after the pre-processor stage. To do so, one enters:

```
gcc -E party.c | wc -l
```

This yields over *seven hundred lines*, for the example touches the standard output, and the compiler needs to link to the required source and to properly parse the file. The impact is apparent by the extensive addition of function and constant definitions.

By contrast this code without the linked module:

```
int main(){
    return 0;
}
```

Produces minimal pre-processing output:

```
# 1 "bare.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "bare.c"
int main(){
 return 0;
}
```

Frequently, in embedded applications on a microcontroller [2] one never produces string output – or even interfaces with an operating system. In these situations, the *standard* libraries may serve no purpose.

[2] http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html

To inspect the lower level code the compiler produces, one must include the appropriate flags. The `-S` option tells the compiler to run the preprocessor, optimize, and assemble the file into a `.s` version.

```
gcc -S party.c
```

```
   .file "party.c"
   .text
   .section .rodata
.LC0:
   .string "Party Cannon!"
   .text
   .globl  main
   .type main, @function
main:
.LFB0:
   .cfi_startproc
   pushq %rbp
   .cfi_def_cfa_offset 16
   .cfi_offset 6, -16
   movq %rsp, %rbp
   .cfi_def_cfa_register 6
   leaq .LC0(%rip), %rdi
   movl $0, %eax
   call printf@PLT
   movl $0, %eax
   popq %rbp
   .cfi_def_cfa 7, 8
   ret
   .cfi_endproc
.LFE0:
   .size main, .-main
   .ident  "GCC: (Debian 8.2.0-20) 8.2.0"
   .section .note.GNU-stack,"",@progbits
```

Chapter 7

Data Types

The C language uses *manifest typing*, so developers must declare all variable types before using them.

The C language uses *static typing*, so variables cannot change their type within the same *scope*.

7.1 Variable Names

For the GNU coding standard: `https://www.gnu.org/prep/standards/html_node/Writing-C.html`

Rules for symbol naming:

- Permitted: Underscore
- Permitted: Alpha-numeric
- Forbidden: Blank spaces, commas
- Forbidden: Keyword or reserved word
- Forbidden: Leading number

Naming convention dictates global variables use long, descriptive names with meaning. Local variables, with limited scope, may use terse names, for their use should be easier to discern in the smaller space.

Avoid using abbreviations in variable names, for they only hold meaning when the project is under active development. As it lingers, people forget the names, and the code suffers.

```
/* A descriptive variable meeting these standards */
int num_active_users = 0;

/* Something suited for another language */
int numActiveUsers = 0;
```

7.2 Primitive Data Types

When developers specify variable types in the source file, they provide useful information to both the compiler as well as other humans. In C, the type specifier instructs the compiler about how much memory it should reserve to hold the data contents.

For maximum memory efficiency, C developers frequently select variables that *precisely* match the minimum and maximum ranges for the abstraction. A boolean value, true or false, only requires a single bit. Consequently, selecting a long double to track this fact – a data type requiring *10 bytes*, wastes memory.

Type	Size (bytes)	Value Range
signed char	1	-128 to 127
unsigned char	1	0 to 255
int	2	-32,768 to 32,767
unsigned int	2	0 to 65,535
short	2	-32,768 to 32,767
long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
float	4	1.2E-38 to 3.4E+38 (6 decimal places)
double	8	2.3E-308 to 1.7E+308 (15 decimal places)
long double	10	3.4E-4932 to 1.1E+4932 (19 decimal places)

Casting

C provides developers a means for treating variables like different types, for everything in the stream is simply a 1 or 0 and it's only through our interpretation that the variables get meaning. Should the number 0 represent nothing, or false?

```c
#include <stdio.h>

short iValue = 320;
char cValue = 20;
double dValue = 3.20;

int main(){
   printf("\niValue: %05d (short)", (short)iValue);
   printf("\n      %5d (char) [Overflow]", (char)iValue);
   printf("\n%13c (ASCII)", (char)iValue);
   printf("\n     %04.2f (float)", (float)iValue);

   printf("\n----------------\n");
   printf("\ncValue: %05d (short)", (short)cValue);
   printf("\n      %5d (char)", (char)cValue);
   printf("\n%14c (ASCII)", (char)cValue);
   printf("\n      %04.2f (float)", (float)cValue);

   printf("\n----------------\n");
   printf("\ndValue: %05d (short) [Precision Loss]", (short)dValue);
   printf("\n      %5d (char) [Precision Loss]", (char)dValue);
   printf("\n%14c (ASCII)", (char)dValue);
   printf("\n       %04.2f (float)", (float)dValue);
   printf("\nComplete.");
}
```

Formatting Variables

When displaying, writing, or working with data, one must sometimes convert the data to a string representation. When doing so, these special characters help the text appear as the developer intends.

Code	Formatting Option
%d	decimal integer
%6d	decimal integer, at least 6 wide
%f	floating point
%6f	floating point at least 6 wide
%.2f	floating point, two characters after decimal
%6.2f	floating point, at least 6 wide and 2 after decimal
%6.02f	floating point, at least six wide, two after decimal include zeros
%o	octal
%x	hexidecimal
%c	character

7.3 Other Data Types

7.3.1 Enumerations

```
/* The General form appears as: */
enum flag { const1, const2, ..., constN };

/* Applied to a basic example */
enum heading {
   north = 0,
   east = 9,
   south = 18,
   west = 27,
};

/* GNU Coding encourages use of enumeration for integer constant
    values */
#DEFINE INSTEAD_OF_THIS  40
```

An Intelligent Example

Although the C language does not support boolean primitive values, one can define values for `true` and `false` through an enumeration. This can significantly improve code readiblity.

```
#include <stdio.h>

/* Including boolean support when it's not built-in */
enum boolean { false, true };

int main(){
   printf("Value of false: %d", false);
   printf("Value of true: %d", true);
}
```

7.3.2 Arrays

A *derived* data type. The Array represents a collection of like type values. Similar to Python, Arrays in C use the square bracket notation for index values. Indexing begins, correctly, at zero. Unlike Python, C forbids negative index positions.

```
#include <stdio.h>
int scores[15];

int main(){
  for(int index = 0; index < 15; index++)
    printf("Player %02d: %5d\n", index, scores[index]);

  return 0;
}
```

One may also declare multi-dimensional arrays in c.

```
#include <stdio.h>

int main(){

  /* One dimensional */
  int vals[] = {10,15,20};
  /* Two dimensional */
  int matrix[2][2] = {{1,2},{3,4}};
  /* Three dimensional */
  int image[][][];

  printf("%02d %02d %02d\n", vals[0], vals[1], vals[2]);

  printf("===================i\n");
  for( int rowCounter=0; rowCounter < 2; rowCounter++ ){
    for( int colCounter=0; colCounter < 2; colCounter++)
      printf("%02d ", matrix[rowCounter][colCounter]);
    printf("\n");
  }
  return 0;
}
```

Pointers: A look Ahead

Frequently, C developers convert between arrays of different dimension types. To do so, one may make use of pointers and casting. In C, the base variable name for all arrays references the memory address on the computer for the physical data location.

```
#include <stdio.h>

int main(){

  /* Two dimensional */
  int matrix[2][2] = {{1,2},{3,4}};

  /* Converted into a one dimensional */
  int *one_dim;
  one_dim = (int*)matrix;

  /* Displayed through pointer referencing */
  printf("%02d %02d %02d %02d\n", *one_dim, *(one_dim++),
      *(one_dim++), *(one_dim++));
  return 0;
}
```

7.3.3 Structures and Unions

Structures and unions both provide a mechanism for grouping together like fields. They differ significantly in their memory requirements. Every element in a struct possesses a unique memory location to hold its contents. The members in a union, however, share the SAME memory location.

```
#include <stdio.h>

union thing_one{
  int number;
  char values[2];
};

struct thing_two{
  int number;
  char values[2];
};

int main(){

  union thing_one first;
  struct thing_two second;

  printf("\nsizeof(union) : %2d",sizeof(first));
  printf("\nsizeof(struct): %2d",sizeof(second));
  return 0;
}
```

Structures are valid return types, function parameters and variables. Structures allow developers to group together tightly related fields into a single container.

One may access structure members through dot notation.

7.3.4 Void

For expressions with an irrelevant value, C supports the `void` data type. Of note, `void` requires the same number of address bytes as any other address, so one may use it as a placeholder for unknown values.

Frequently `void` appears as a return type for a function.

Chapter 8

Statements, Operations, and Expressions

An expression in C becomes a statement when followed by a semicolon. Braces allow developers to create both *blocks* and *compound statements*. The main difference between the two is that blocks of code include a variable declaration. Without the declaration, it is simply a compound statement.

Unlike Python, C ignores indentation when parsing the file to verify it meets the language syntax. C requires a semi-colon as a terminator for every statement, and the semicolon, by itself, is a valid bit of c. Consequently, this is valid [1]:

```c
#include<stdio.h>
int main(){
;
;;
;;;printf("\nseriously?!\n");;;;;return 0;}
```

8.1 Conditional

C implements `if` and `else` as one might expect from a programming language developed after the moon landings. Generally, the if-else statement takes the form:

```c
if (expression)
```

[1]But horrible. Don't do this. Just because you can do a thing does not mean you should do a thing

```
  statement;
else
  statement;
```

The else portion remains optional. Unlike Python, C lacks a special `elseif` conditional, but one may chain if statements together to achieve the same result.

8.2 Switch

When a conditional variable may only take an limited number of constant, integer arguments, the `switch` statement provides an efficient mechanism for multiple-path decision making. For example, a streetlight typically takes Red, Green, and Yellow values. Rather than structuring a series of *if-else* statements, the code might benefit from a `switch`.

The case statement imposes a few restrictions while offering offers unique behaviors:

1. Every case condition must be unique – no identical values
2. C only executes the `default` statement if nothing else fired
3. Cases, including the `default`, can appear in any order
4. The `break` statement terminates the switch evaluation
5. The `switch` will *fall-through* to the next statement unless it encounters a `break`

8.3 Operators

8.3.1 Arithmetic Operators

C supports basic arithmetic operations with their logical meanings. These are: `+,-,*,/, and %`

One may apply these in interesting ways. Recall that the characters printed to the screen representing mappings of numbers to a graphical depiction. One may represent the first character in the english alphabet with a letter symbol or with a number. The ASCII character codes map the symbols we see displayed on the screen to their numeric equivalents.

Consequently, characters may logically be manipulated with arithmetic operations:

2

Figure 8.1: The Nintendo Switch is unrelated to C-style switch statements, but it likely includes them in its operating system. Bethesda's award-winning game *Skyrim* also includes switches in many puzzles – equally unrelated.

```
#include <stdio.h>

int main(){
  printf("%d\n", 'a' + 'A');
  printf("%d\n", 'a' - 'A');

  return 0;
}
```

8.3.2 Relational and Logical Operators

Relational operators: `<, <=, >, and >=`

Equivalence operators: `== and !=`

Logical operators: `&& and ||`

Logical operators link expressions to establish the truth or falsehood of a statement. C evaluates the expressions connected by the symbols from the left hand side to the right hand side. The checking stops the moment C establishes its value. That is, if the statement on the left fails, and the two statements are connected by an and symbol, then the program need not continue evaluating terms on the right. Alternatively, when connected by an or statement, if the left side evaluates true, it need not continue evaluating terms.

```
#include <stdio.h>
```

```
int main(){
  int count = 0;
  if( ++count || ++count || ++count || ++count )
    printf("Done checking: %d", count);
  return 0;
}
```

8.3.3 Increment and Decrement

C includes two operators, `++` and `--` which increments and decrements the variable as appropriate. Their position with relation to the variable plays a subtle, but significant role.

```
#include <stdio.h>

int main(){
   int tally = 0;

   printf("Current: %d\n", tally);
   printf("Post increment %d\n",tally++);
   printf("Current: %d\n", tally);
   printf("======================\n");

   printf("Current: %d\n", tally);
   printf("Pre increment %d\n",++tally);
   printf("Current: %d\n", tally);

   return 0;
}
```

How about this fun:

```
#include <stdio.h>

int main(){
  char alpha;
  char bravo;

  alpha = 1;
  bravo = (alpha = ++alpha);

  printf("Value of Bravo: %d", bravo);
  return 0;
}
```

Thankfully, this is invalid, for the expression `bravo++` cannot appear on the *left side* of an assignment statement.

```
/* Not valid syntax, bravo++ is not a valid l-value */
bravo++ = ++bravo;
```

What does this print (OH SNAP):

```
#include <stdio.h>

int main(){
  char bravo = 1;

  printf("Value of Bravo: %d", bravo = (bravo = bravo++));
  return 0;
}
```

8.3.4 Bit-wise

Bit operators manipulate the actual 1 and 0 values within a larger primitive like a `char` or `long` independent of it's sign value. These operations do not, however, work with the floating point abstractions.

Operator	Description
&	Bitwise AND
|	Bitwiwse OR
^	Bitwise NOT
«	Left shift
»	Right shift
~	One's complement

Frequently, one uses bit operations to toggle the pins of a micro-controller. For example:

```
/* Laser on pin 3 */
#define NOT_LASER 0x04

int fire_laser(){
  /* Make sure it's high first */
  GPIO1 |= NOT_LASER;
  /* Pull low to fire */
  GPIO1 ^= NOT_LASER;
  /* Return high to turn OFF */
  GPIO1 |= NOT_LASER;
}
```

Chapter 9

Pointers and Memory

In addition to the primitive types discussed, types which contain data with meaning to the program, C supports a variable type specific to holding addresses. That is, they hold the meta-information about where on memory the actual data exist so the machine knows where to find them during its calculation.

9.1 Addresses

In C, Java, and C++ (among many others), the language includes a way for discussing where the data are located in memory. As a system level programming language, C programmers frequently interact with the machine at the memory level, so this functionality becomes increasingly useful. Every variable, even primitive variables, reside somewhere in memory. To access this in C, one uses the following syntax:

```
#include <stdio.h>

int main(int argc, char* argv){

  unsigned short ex1 = 0xBBBB;

  printf(" Value of variable: %8X\n", ex1);
  printf("Address of variable: %8X\n", &ex1);
  return 0;
}
```

Which could return something like:

```
 Value of variable:  BBBB
Address of variable:  A6D6B32E
```

The user's context is the value of the variable, and the address represents it location on the machine. On a 64-bit processor, the address requires 64 bits regardless of the actual data type.

```
#include <stdio.h>

int main(int argc, char* argv){

  unsigned char ucEx = 'a';
  unsigned short usEx = 0xBBBB;
  unsigned long ulEx = 0xBBBBBBBBBBBBBBBB;
  float        fEx = 3.5;

  printf("Address of ucEx: %8X\n", &ucEx);
  printf("Address of usEx: %8X\n", &usEx);
  printf("Address of usEx: %8X\n", &ulEx);
  printf("Address of fEx: %8X\n", &fEx );
  return 0;
}
```

Which could produce:

```
Address of ucEx:  280D848F
Address of usEx:  280D848C
Address of usEx:  280D8480
Address of fEx:  280D847C
```

Observe that the address seems uncorrelated with the data contents. That is, nothing about the address of the unsigned character `ucEx` indicates what value it holds. The same follows for each of the other variables.

Also note, the addresses, although unrelated to the underlying data, appear closely related. This is because the machine allocated space for them *on the stack* when it entered the main function. Naturally, these addresses appear sequentially because that is how the computer stores them as it performs a function call.

The variables introduced as parameters in a function call, as well as those it declares in its body, all appear on the *stack*. One may also declare variables on the *heap*. It obviously includes a different range of addresses:

```
#include <stdio.h>
#include <stdlib.h>
```

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

  int *first, *second;

  /* dynamic allocation of space for one integer */
  first = (int*)malloc(sizeof(int));
  second = (int*)malloc(sizeof(int));

  *first = 1;
  *second = 2;

  printf("Address of first: %X\n", &first);
  printf("Address of second: %X\n", &second);
  printf("------------------\n");
  printf("Address pointed to by first: %X\n", first);
  printf("Address pointed to by second: %X\n", second);
  printf("------------------\n");

  /* Illustrate two ways to access contents of variable */
  printf("Contents of first: %d\n", first[0]);
  printf("Contents of second: %d\n", *second);

  /* Oh no! Memory leak! */
  return 0;
}
```

Which could produce output:

```
shawn@raider: $ ./a.out
Address of first: 177A8B78
Address of second: 177A8B70
------------------
Address pointed to by first: 74BF6260
Address pointed to by second: 74BF6280
------------------
Contents of first: 1
Contents of second: 2
```

Note the value disparity between the addresses pointed to on the stack and those that appear on the heap. The heap allocations, in this case, appear near each other, but there exists no requirement for this. Significantly, their addresses are in a significantly different region of space – top vs. bottom of memory.

Also note, the above application fails to free the allocated memory after execution. C lacks a garbage collector like Java and Python. Consequently, this leaves those data allocated on the heap until the machine reboots.

9.2 Pointers

As the above examples illustrated, in addition to inspecting the address of primitive data types, one may store these data in their own type. The *pointer* is a data type that holds the address to another variable, and like other variables, one may perform operations on them and include them as function arguments and return values.

Java also includes a way to store addresses. In fact, the mechanism is so embedded in the language that new developers can go whole semesters before encountering a `NullPointerException` while implementing a linked list. The pointer also appears when Java developers print their first new classes and, instead of seeing their data as expected, they encounter a stream of hexadecimal characters.

In addition to potentially pointing to addresses for variables in stack memory, the pointer may hold an address to a variable outside the stack and deep within the heap area of memory. This allows for many useful computer science abstractions. In fact, pointers may be one of the most significant introductions to programming and software design, for they offer overwhelming power.

9.2.1 Dereferencing

To access the contents of the address pointed to by a pointer, one may use multiple forms. Bracket notation, previously used with arrays, may be used to project into a pointer array, or one may use the `*` asterisk [1]

```c
char *data = "This is it.";

data[2];
*(data+2);
```

Pointing into structures

Structures allow developers to cleanly group together related fields. One can create a structure for the player's ship, and in this structure it might hold

[1] Or *Kleene Star*

several related fields. What if one wished to implement a star-ship dealership where the user scrolled through the dealership browsing for available craft? How might one store the currently selected ship?

Moreover, how might we assign the selected ship to a player? Store an index to a greater array? After selecting the ship, the user might wish to customize it with additional cargo holds or superior engines. Consequently, creating a data type with a link to another data type might be in order.

In any event, pointers to structures appear frequently within C code. Fortunately, C includes special syntax for accessing data members inside a structure pointed to by a variable. The following example illustrates both ways.

```
#include <stdio.h>
#include <stdlib.h>

struct entry {
  char *key;
  int count;
};

struct node {
  struct entry *data;
  struct node *next;
};

int main(){

  struct entry *player = (struct entry*)malloc(1);
  player->key="Mr. Jones";
  player->count=1;

  struct node *head = (struct node*)malloc(1);
  head->data = player;

  printf("First player %s\n", head->data->key);

  /* return the heap memory to the OS */
  free(player);
  free(head);

  return 0;
}
```

9.3 Pointers and Arrays

Although introduced earlier, arrays remain closely intermingled with pointers. In fact, the differences remain subtle especially when inspecting the notation. For example:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv){

  /* declare a consecutive block of objects on the stack */
  int first[10] = {5, 6, 7};

  /* declare an address to an integer on the stack */
  int *ptrSecond;
  /* point it to the second item in the list */
  ptrSecond = &first[1];

  printf("Contents: %d followed by %d\n", ptrSecond[0],
      ptrSecond[1]);
  return 0;
}
```

In the Array and pointer access example, the program would first print the contents of the address pointed to by `ptrSecond` (which is the second item in the first list) as well as the contents of the bin one slot away from this base: 6 and 7.

One may also perform arithmetic operations on pointers.

```
#include <stdio.h>
#include <stdlib.h>

int main(){

  /* Strings are just pointers to characters */
  char *greeting = "Wake up. Time to die.";
  /* One may perform math on pointers */
  greeting += 7;

  while( *greeting ){
     printf("%c",*greeting--);
  }
  printf("\n");
  return 0;
}
```

What about the following insanity? It takes advantage of the fact that strings in C are *null-terminated* and each of the input arguments to the function are pointers. That is, they point to fixed locations directly in memory.

```
void what_is_this( char *source, char *target){
  while(*source++ = *target++);
}
```

This code takes advantage of the fact that assignment statements are *expressions* in C, so setting a variable to a value produces an actionable result. The `while` statement continues until evaluating a NULL character.

9.4 Pointers to Pointers

Multi-dimensional arrays serve as an example of when one would want a pointer to a pointer. Having already established the relationship between an array variable's name and its address, we know we can assign pointers to this address.

Given that an array, which points to a consecutive block of memory, can contain any value appropriate for the machine architecture, it can also hold, within those array cells, addresses of other arrays. Inception style, each of these arrays may, themselves, contain pointers to other arrays.

Are strings not pointers to characters? An array of strings is an array to character addresses. There are subtle differences between the two as this snippet illustrates:

```
int img[512][512];
int *cpy[512];
```

In both examples, C permits the developer to reference the contents of these items using bracket notation:

```
cpy[10][10] = img[10][10];
```

Both examples point to memory locations, but in a multi-dimensional array, when declared using the bracket notation, the machine reserves enough *consecutive* space to hold the data. For the pointer definition, the original cpy declaration reserved space for 512 pointers, but it did not actually point these cells toward something meaningful.

In the second example, nothing says the item pointed to at cpy[10][10] should be anywhere near the item located at cpy[9][9] like it would with a proper array declaration.

9.5 Dynamic Allocation

Arrays rarely precisely match their underlying contents. As we have seen with the address allocation, the system reserves space for arrays declared in a function on the stack, so changing the size later becomes problematic, for other variables are both in front and behind the array.

In some situations, developers know the size of the array before execution, but these opportunities seem limited. Instead, a more efficient memory use model would allow for functions and programs to request *precisely* the amount of memory they need based upon the current execution environment.

Why allocate an array for 10,000 players if only fifty use the service on a typical afternoon? Instead, store the data in a data structure that precisely matches the contents. Pointers facilitate this behavior, for with them we construct linked lists, dictionaries, and dynamic arrays.

Rather than storing an actual array of 10,000 strings, store a pointer to an array, and then allocate space for the array at run-time based on the number of users.

Allocating Memory

The functions malloc and calloc, available through stdlib.h provides access to dynamic memory tools.

The standard library function calloc differs from malloc in that it initializes the memory area to zero before returning it. Malloc may leave behind the remains of previous data. That is, the memory area returned by malloc may already have been used and returned by the program, so it might contain previous, and potentially private, data.

Malloc allocates space in *bytes*.

```
int *ptrFellowship = malloc(7 * sizeof (int));
if(ptrFellowship == NULL) {
  /* Memory could not be allocated */
  exit(EXIT_FAILURE);
```

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv){

  int *to_do;

  /* Make original space for 10 */
  to_do = (int*)malloc(sizeof(10));
  printf("Printing unitialized contents: %d\n", to_do[9]);

  /* decide to increase the array to 100 */
  free(to_do);
  to_do = (int*)calloc(sizeof(100));
  printf("Printing unitialized contents: %d\n", to_do[99]);

  return 0;
}
```

Figure 9.1: Can you spot the memory leak?

```
}
```

Calloc allocates space for `n` blocks of some size. It also asserts the memory space it provides contains pure zeros, so one will not need to perform additional initialization. Due to the initialization step, `calloc` requires more execution time.

```
int *ptrFellowship = calloc(7,sizeof(int));
if(ptrFellowship == NULL) {
   /* Memory could not be allocated */
   exit(EXIT_FAILURE);
}
```

Freeing Memory

A significant source of memory leaks in low-level, system programs – developers must make sure memory allocated on the *heap* returns to the operating system for use in other applications. Java and other languages include a *garbage collector* that comes along behind the scenes and liberates unused memory.

This mechanism, however, requires processor cycles from the JVM, and C lacks a JVM performing assembly instructions behind the scenes. After receiving a memory allocation, the program *must* return this memory to the operating system after use. To do so, simply make use of the `free` function. It takes the variable to liberate as a parameter.

Chapter 10

Functions

Functions help break up complicated and repetitive tasks into precision subroutines. The `main` function appearing in many of the code examples to date illustrates one function. Although C mandates specific treatment for the function named `main` it does permit developers to create and use any other number of functions.

10.1 Function Declaration

As a *block-structured* language, C allows developers to declare variables in *blocks* of code. C, unlike some block-structured languages, forbids the definition of other functions within a block. Thus, one cannot define a new function while within main. Instead, C developers must create a new function at the same level.

Furthermore, before using a function, C must understand what to expect back from the function as well as what parameters it expects loaded on its stack for use. For trivial programs, with only one function, simply implementing the function 'north' of the call in the source file fulfills this obligation. Because C already observed the function, it understands what parameters it needs to operate when `main` calls it later. When one begins to expand to real tasks, however, this becomes impractical. Functions themselves frequently refer to other functions, so requiring the total implementation before use establishes a mystifying function order in the source file where simply rearranging the code causes compilation to fail.

Instead, C allows developers to simply supply some quick notes about what the function needs in order to establish how to use it. As long as the source file declares the function before using it – not necessarily implementing it in code before use – the file will compile. To do this, developers must first *declare* a function by providing its signature or prototype.

For example [10]:

```
#include<stdlib.h>
#include<stdio.h>

/* declare it so main knows how to prepare for it */
int get_line(char dest[], int limit);

int main(){
  char from_user[80];
  printf("%s\n%d long!", from_user, get_line(from_user, 80));
}

/* Now include the actual code */
int get_line(char dest[], int limit){
  int cur;
  int index;

  for( index = 0; index < limit - 1 && (cur=getchar())!=EOF &&
      cur!='\n'; ++index)
    dest[index] = cur;
  if( cur == '\n' ){
    dest[index] = '\n';
    index++;
  }
  dest[index] = '\0';
  return index;
}
```

After calling this function, the display might produce:

```
shawn@raider: $ ./a.out
San Diego State University
San Diego State University
27 long!
```

C provides access to functions declared in other source modules through the `include <`*`file.h`*`>` and `extern` mechanisms. The examples provided in this document make extensive use of functions declared in other modules, for the `printf` function declaration lives in `stdio.h`.

Pedantic Note: A *function signature* allows the compiler to resolve names and overload functions as needed. Many functions may share the name `print`, so C uses not only the function name, but its parameter types when deciding which function to use. A function signature does not include its return type. On the other hand, a `function prototype` includes the name and its parameters, but it also includes the return type.

10.2 Parameters and Return Values

Functions take formal *parameters*, or arguments, which they they may use during the course of their execution. Prior to making the function call, the machine loads the *values* for each of these parameters on the stack. Each function call thus begins with a fresh copy of the relevant data.

In the case of recursion, this becomes critical, for the recursive calls receive their own copies of the variables which they cannot change. Thus, one cannot perform a swap using value parameters.

```
/* Consumes energy and time */
void swap(int a, int b){
  int temp = a;
  a = b;
  b = temp;
}
```

Instead of modifying the values themselves, however, one might implement the function with pointers.

```
/* It's a swap! */
void swap(int *a, int *b){
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

After completion, functions `return` a value to the caller. We have seen `int` used frequently with the `main` routine, but functions can return any data type. Recall that `malloc` and `calloc` each return the address of a `void *` the caller must cast into the appropriate data type.

Some functions need not produce a meaningful return value. For example, a function that prints instructions – which simply executes a series of statements – likely need not return a value. What type would it be? What would it mean?

At the same time, one might wish to create a function which initializes a player structure to some series of values based on some key inputs (e.g., character class, strength, comeliness, charisma). This style of builder function might return a pointer to the newly created player. This flexibility allows developers to create clearer code.

10.2.1 Variable Parameters

Because C overloads functions using the method signature, one function may require multiple implementations in the software. In many situations, this presents minimal trouble. Reasonably, however, situations present themselves where the software developer might expect any number of arguments without impacting the algorithm.

For example, the command line arguments supplied to many UNIX shell programs take a variable number of input arguments. One might include a `-last` flag when printing the contents of a directory, or one may skip this flag. The function itself simply keeps parsing the input arguments until they complete or stop making sense.

1. point to the first argument
2. verify it is valid
3. save parameter for program use
4. point to next argument in list
5. return to 2 until arguments empty

The `main` function stands out as the quickest variadic function to explore. It includes two parameters when linked to the operating system. The first establishes the number of space-delineated input arguments, and the second serves as a pointer to an area in memory containing each of those arguments. [1]

The user might supply any number of arguments to the program, but doing so would create a nightmare scenario if software developers needed to anticipate each possible entry combination with a function signature. Instead, C allows developers to create functions using a pointer to the input arguments.

Generally, C includes a header file with functions for accessing variadic parameters. Again, `main` represents a special case, but any function can

[1] This format illustrates a *specific* variadic implementation for main. Because developers frequently include command line arguments with main, C creates the necessary hooks for it automatically.

include optional parameters. For these, one must use a different method for extracting the parameters.

Declaration Syntax

When declaring a function with a variable number of input arguments, just as in Python, the variable parameter must appear as the last item in the argument list.

```
void func (char *a, int b, ...){
  /* Do something */
}
```

This function requires two mandatory arguments, `a` and `b`, as well as any number of optional arguments. Here, we see the difference between this definition and the `main` definition.

```
int main( int argc, char *argc ){return 0;}
```

This method clearly defines the second argument, and both are *required*. When the user launches the program at the command line, the program automatically extracts and computes these values from the keyboard. Because the variable input arguments appear as a character pointer – only here in main – we can access them through array indexing.

Using the Argument Values

Normal variables require a type specification as well as a name, but the variadic function argument appears anonymously as `...`. How does one access the second item in this list? The answer appears somewhat painfully.

To do so, one must use a special type defined in `stdarg.h`: `va_list`. To initialize this variable, the code must call `va_start`. After completing the call, the developer may begin reading the arguments by calling `va-arg`.

The following example (`https://www.gnu.org/software/libc/manual/html_node/Variadic-Example.html#Variadic-Example` illustrates their complete use:

```

#include <stdarg.h>
#include <stdio.h>

int add_em_up (int count,...){
  va_list ap;
  int i, sum;

  /* Initialize the argument list. */
  va_start (ap, count);

  sum = 0;
  for (i = 0; i < count; i++){
    /* Get the next argument value. */
    sum += va_arg (ap, int);
  }

  /* Clean up. */
  va_end (ap);
  return sum;
}

int main (void) {
  /* This call prints 16. */
  printf ("%d\n", add_em_up (3, 5, 5, 6));

  /* This call prints 55. */
  printf ("%d\n", add_em_up (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

  return 0;
}
```

10.3 Variable Initialization

Typically, programs include data acquired dynamically at run time as well as working and scratch-pad values the program encounters during execution. The `swap` function, for example, requires a temporary variable holding the first value replaced.

Variables must frequently begin use from a known starting state. A `for` loop, for example, expects to initialize the index counter to a starting value. When does it initialize this value if nested within another `for` loop?

In block-structured languages like C, the program initializes these `automatic` and `register` values when the program first enters the current *block*. Thus, in the nested `for` loop example, each time the loop first starts, C initializes the variables.

For externally defined and static variables, C initializes these to zero (if undefined) before program execution begins. C makes no promises about the uninitialized state of other variables.

```
#include<stdio.h>

static int everywhere;

int main(){
  int value;
  int *pointer;
  float superfun;

  printf("%d\n%d\n%X\n%f\n", everywhere, value, *pointer, superfun);
  return 0;
}
```

In this example, the integer `value` may print any value within the valid range for the type on the machine. On the other hand, even though `everywhere` also uses an integer type, the code declares it as a `static` variable. Consequently, the program automatically initializes it to zero.

10.4 Functions as Formal Parameters

Because functions exist in memory, C allows developers to point to their memory locations. That is, one may create pointers to functions. More significantly, one may then execute these functions. In this fashion, one might begin including functions themselves as input arguments to other functions.

One example of this might be establishing a function to call when the system observes a left mouse button activation. For roughly ninety-percent of the population, this action activates the primary mouse command. Some users, however, map this action to the *secondary* action and use the right mouse button as its primary input instead.

Instead of creating two functions for this action, the program simply implements a single mouse button callback that accepts the function it should call when it detects the event. For some users, it will call the primary mouse button function, but others will map this to a different action.

The following code fragment declares a function named `myfunction` which returns `void` and accepts a single `void` pointer argument. This function argument, however, itself imposes the restriction that it takes a single argument. Thus, we declare a function`myfunction` which takes as its only argument a pointer to a function which takes an integer as an argument.

```
void myfunction(void (*fptr)(int));
```

An example of this in practice should help clarify the situation [11].

```
#include <stdio.h>

void print()
{
    printf("Hello World!");
}

/* A toy function accepting a pointer to a (void)(void) */
void helloworld(void (*fptr)());

int main(void)
{
    helloworld(print);
    return (0);
}

void helloworld(void (*fptr)())
{
    /* Execute the function the caller provided */
    fptr();
}
```

Or more Usefully

```
#include <stdio.h>

void printNumber(int nbr)
{
    printf("%d\n", nbr);
}

void myFunction(void (*f)(int))
{
    for(int i = 0; i < 5; i++)
    {
        (*f)(i);
    }
}

int main(void)
```

```
{
   myFunction(printNumber);
   return (0);
}
```

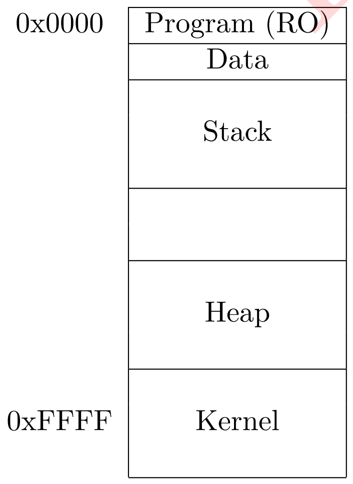
10.5 Stack Smashing

One of the earliest *buffer overflow* attacks targets the parameter stack. The attacker attempts to override a buffer with too much precisely-crafted, malicious-input. If successful, the attack overwrites return values on the stack and allows the attacker to inject code into the system.

Recall that when the computer prepares to execute a function call, it first pushes its own state, along with each of the function's parameters, onto the stack. Moreover, variables declared and used within the scope of the function also appear on the stack. Consequently, blowing through the bounds of any of these can lead to arbitrary code execution on the stack.

10.5.1 The Computer Stack

A stack is a contiguous block of memory containing data. A register called the stack pointer (SP) points to the top of the stack. The bottom of the stack is at a fixed address. Its size is dynamically adjusted by the kernel at run time. The CPU implements instructions to PUSH onto and POP off of the stack. [12]

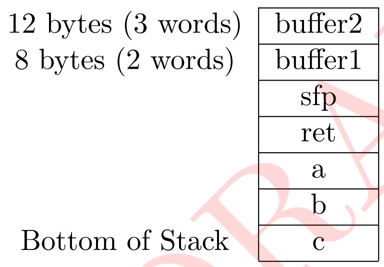


10.5.2 A Stack Frame

> The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call.[12]

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}

void main() {
  function(1,2,3);
}
```



10.6 Buffer Overflow

This attack attempts to overwhelm the *stack* where the computer stores return addresses. If one can overwrite a return address with the address of malicious code[2], one may take control of the computer. Perhaps the buffer overflow attack includes a simple function to access the shell?

Because the program permits unconstrained buffer writes, for it uses **strcpy**, it will continually read in data even if the container it attempts to store them in overfills.

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
```

[2]Friendly to you though.

```
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

Many C compilers now warn the developer when the code module includes access to `gets` or `strcpy` due to their vulnerabilities. That said, they remain a part of the language to support backward compatibility as well as ... stunt pilot.

10.7 Exercises

1. Write a program which returns the number of capital letters in an input string of characters.
2. Create a function that determines if two, null-terminated input strings differ by a single character.
3. Given an integer, write a function to return its Roman numerial equivalent (only consider the range from 1 to 3999). *Hint: Consider building up the Roman numerial string from greatest to smallest.*
4. Write a function which determines if the two input characters differ by a single bit.
5. Produce a function that accepts two arrays, each already sorted, and returns a new combined array with the elements in sorted order.
6. Zero-sum game: Given an input array of signed integers, produce a sub-array containing only those values which, when summed, produce zero. If no solution exists, return an empty array.
7. *Difficult:* Given an array of item values and an equally sized array containing their corresponding weights (expressed as an integer), write a function (possibly recursively) that returns the maximum possible combined value of items from the list without surpassing an arbitrary

but specific weight limit. Consider loading a player character up with the maximum amount of loot before becoming encumbered.

8. *Difficult* Given a two dimensional array containing integer costs, compute the path from the upper-left cell to the lower-right cell with the **greatest** cost. As a simplification, the player may only move to the right or down, so the number of possible paths remains finite and the player cannot create any cycles.

Chapter 11

The C Preprocessor

The C programming language supports *pre-processing* source files. The pre-processor performs helpful, syntactic substitutions to improve the readability of source code. It prepares the source file for lexical analysis without generating any compiled units. One may think of the output of the C pre-processor as more source code in a more computer-readable form. The pre-processor removes comments, for example, which assist humans but offer the compiler nothing.

The C language provides *header* files to assist the compilation process. By convention, C header files end with the extension `.h`. These files help the compiler, but they are not *themselves* compiled. In truth, one may use any valid file name here, but using something other than the standard for a header file seems unnaturally obtuse.

To communicate with the pre-processor, one begins the line in the file with the `#` symbol.

11.1 Including Files

Often times, one needs to import definitions and functions created earlier or by other developers. The examples in this document frequently include samples of this behavior with the `printf` and `malloc` functions. This need not apply only to those functions created and included in the standard compiler installation, for one may also include references to custom *header* files the developer creates specifically for the task at hand.

Developers may specify the included file using either <*filename.h*>or by including the desired file in quotation marks as the following example illustrates.

```
#include <stdlib.h>
#include "node.h"
```

The `include` pre-processor command instructs the pre-processor to insert the contents of the indicated file at the point of reference. Because this is a simple text substitution – the contents of the file are pasted into the source file where the statement appears – the order of include statements may impact the program's operation.

If two files each define conflicting values for the boolean `TRUE`, it could impact the performance of some of the loops and statements within dependent functions. Fortunately, the C-pre-processor includes instructions to help prevent some unintended consequences. For example, what happens when two header files both include the same additional dependency, for header files themselves may contain references to other header files.

In fact, good software design encourages modular code, so multiple include statements becomes the norm. Some large projects use a single project-wide include statement which itself contains include statements to other files. By convention, include statements appear at the top of a c source file. The order of the statements impacts the order of definitions; the coding standard in use for a particular project typically dictates the order and format of include statements (e.g., library files in angle brackets and custom ones in quotes").

```
#include <somestandardlibrary.h>
#include "mycustom.h"
```

11.1.1 External Variables

The C language supports using variables defined in other code modules. These *global variables* are accessible through some additional syntax. The keyword `extern` instructs the compiler that the variable definition that follows serves as a placeholder for the actual variable. These *external* variables appear somewhere else in the software package. It might be another file, or it could mean simply *external to the current function scope*.

The `extern` keyword serves as a bit of notation for the compiler and linker. The compiler knows how to use the external variable without needing to see it exist first. Then, during the linking process, the compiler fills in the appropriate addresses to the external variables in whatever modules they reside.

11.2 Defining Values

Recall that the `enum` data type provides a neat and efficient method for defining constant integer values. These structures become increasingly useful in larger programs. Good software design encourages developers to establish constant values with a readable description. Why compute the value for Pi every time one needs it when one may simply store an approximation?

```
#define M_PI 3.14159265358979323846
```

Now, rather than using the long float value in the source code, the human developers may use `M_PI` instead. The pre-processor will prepare the code for compilation by substituting out `M_PI` and replacing it with the text following the definition. Be advised, the compiler performs no error checking as to the validity of the replacement.

What were to happen if one defined `M_PI` like so:

```
#define M_PI 3.14159265358979323846;
```

Here, we simply added a single semi-colon at the end of the definition. What are the implications for the source module?

```
/* An assignment statement with the macro substutition */
float raspberry = M_PI;

/* A conditional with the macro substitution*/
if( calculation < M_PI ){
 ...
}
```

Recall that in C, a solitary semi-colon is a valid expression. Consequently, the assignment statement using the macro – which itself includes a semi-colon – inserts an extra statement into the program, but it should not negatively impact the compilation output or program results.

When used in the conditional, however, the compilation breaks, for the extra semi-colon included in the macro's definition also appears everywhere `M_PI` appears. Consequently, after expanding the source code, it becomes:

```
/* An extra statement on the end */
float raspberry = 3.14159265358979323846;;

/* Unbalanced Paren */
if( calculation < 3.14159265358979323846; ){
 ...
}
```

11.2.1 Pre-processor Conditionals

The pre-processor supports a handful of conditional statements one may use to give it meta-instructions about how to conduct the pre-processing. For example, one typically need not include a custom header file in a program more than once, so these files frequently include a conditional band to quickly exit after already inserted once. Imagine a situation where a source file includes two header files, but the second header file also includes the first header file. For example:

```
#ifndef __UNIQUE_HEADER
#define __UNIQUE_HEADER

extern int UNQ_count;

#endif
```

One may observe this behavior by inspecting some of the standard header files included with the operating system.

11.3 Macros

Although functions simplify code, they require additional space on the stack. Alternatively, some methods greatly lend themselves to a macro. That is, instead of calling another function to execute a few lines of code, create a macro with a meaningful name and let the pre-processor do a code substitution before the compiler sees the file.

```
#define identifier token-sequence
#define not_macro because there is a space after the name

#define yes_macro(id-list) token-sequence
```

For example, sometimes one needs a program with an infinite loop. Applications that constantly poll for input and wait, like the *super-main* loop of a simple micro-controller.

```
#ifndef FOREVER()
```

```
#ifndef _STDIO_H

#if !defined __need_FILE && !defined __need___FILE
# define _STDIO_H 1
# include <features.h>

__BEGIN_DECLS

# define __need_size_t
# define __need_NULL
# include <stddef.h>

# include <bits/types.h>
# define __need_FILE
# define __need___FILE
#endif /* Don't need FILE. */


#if !defined __FILE_defined && defined __need_FILE

/* Define outside of namespace so the C++ is happy. */
struct _IO_FILE;

/* ... */
```

Figure 11.1: Sample of some of the `stdio.h` file highlighting use of the `define` and `ifndef` pre-processor instructions. This file works with both C and CPP.

```
#define FOREVER() for(;;)
#endif

int main(){
  FOREVER(){
    /* Poll for input */
    poll_and_process(GPIR);

    /* Put processor to sleep (interrupt will wake) */
    sleep();
  }
}
```

Chapter 12

Parameters

12.1 Applicitive Order

In what sequence does the compiler evaluate the parameters when it prepares for the function call? Does it evaluate them before the call, or only when needed? How does this change with a macro expansion?

12.1.1 Call by Macro Expansion

One implementation of the swap function without the need for a function call. That it, the program flow need not jump into a function to perform the swap and, instead, it takes place explicitly with code.

```
#include <stdio.h>

/* A simple macro for the pre-processor */
#define SWAP(X,Y) {int temp=X; X=Y; Y=temp;}

int main() {
  int a = 3;
  int b = 20;
  printf("a:%d, b:%d\n", a, b);
  SWAP(a, b);
  printf("a:%d, b:%d\n", a, b);
}
```

What happens when a macro defines a variable already in the environment? That is, what happens when a macro defines variables inside its own body? How do these interact with the other variables in the caller? The

```
#include <stdio.h>
#define SWAP(X,Y) {int temp=X; X=Y; Y=temp;}

int main() {
  int a = 2;
  int temp = 17;
  printf("%d, temp = %d\n", a, temp);
  SWAP(a, temp);
  printf("%d, temp = %d\n", a, temp);
}
```

Figure 12.1: An example of *variable capture*. What values for `a` and `temp` does the program print?

```
#include <stdio.h>

int main() {
  int a = 2;
  int temp = 17;
  printf("%d, temp = %d\n", a, temp);
  {int temp=a; a=temp; temp=temp;};
  printf("%d, temp = %d\n", a, temp);
}
```

Figure 12.2: The same swap source code, but without the macro, helps illustrate how the macro captures `temp`.

term *variable capture* describes this behavior, and the code below illustrates its behavior.

Variadic Macros

Like functions, macros also may accept a variable number of input arguments[1].

12.1.2 Pass/Call by Value

The procedure or function receives a copy of the value of the parameter object. Using this style, subroutines may freely modify the value of the

[1] https://gcc.gnu.org/onlinedocs/cpp/Variadic-Macros.html#Variadic-Macros

```
#include <stdio.h>

int res( int first, int second ){
   return (first*first) * second;
}

int main(){

  int a = 2;

  a = res( a++, a++ );

  printf("result: %d\n", a);
}
```

Figure 12.3: C passes by value, but what value and in what order?

parameter objects provided with the call, but these changes in no way impact the value of the variables outside the routine. Incrementing a parameter does not, necessarily, impact any other functions or procedures using the value, for the routine received a copy.

12.1.3 Pass/Call by Reference

One may use pass-by-reference in C or C++ through reference pointers. This causes the parameters to serve as aliases for the outer variable.

12.1.4 Pass/Call by Value-Result

Also known as *copy-restore* – In the event a function's parameter is a reference accessible by something else (another function/routine) it provides a copy of the reference object for the current procedure to use during its execution. When it returns, it updates the original reference object with the changes made during the function call. This way, if other threads simultaneously use the reference object, their concurrent modifications remain totally independent.

Upon terminus, the machine overwrites/merges the original reference object with the changed one. This can lead to unpredictable behavior if multiple threads concurrently modify a shared parameter reference, for their independent modifications may disappear between updates.

Introduced by Fortran.

```c
#include <stdio.h>

int res( int first, int second ){
   return (first+1) * second;
}

int main(){

  int a = 2;

  a = res( res( a++, a++ ), res( a++, a++) );

  printf("result: %d\n", a);
}
```

Figure 12.4: C passes by value, but what value and in what order?

```c
#include <stdio.h>

void swap( int *first, int *second ){
   int tmp = *first;
   *first = *second;
   *second = tmp;
}

int main(){

  int a = 1;
  int b = 2;

  printf("before: a(%d), b(%d)\n", a, b);
  swap(&a, &b);
  printf(" after: a(%d), b(%d)\n", a, b);

  return 0;
}
```

Figure 12.5: Passing by reference in C

12.1.5 Pass/Call by Name

Used by Algol and aspects of it appear in other languages.

Pass in a *symbolic* name.

The argument expression is re-evaluated each time the formal parameter is accessed. The procedure can change the values of variables used in the argument expression and hence change the expression's value. Java's use of Generics – the actual mechanism by which one Instantiates a string array list – uses pass-by-value.

12.1.6 Aliasing

Chapter 13

Object-Oriented Programming Concepts

In order to support *object-oriented programming*, a language must include: abstract data types, inheritance, and dynamic binding. We typically discuss the object-oriented features present in a language, and, based upon those characteristics, then determine if the language *qualifies* as object-oriented. The blurry distinction can lead to aggressive, pointless debate. Languages evolve, and at some point they possess enough features for developers to find them useful when developing in the object-oriented style.

The major features we consider in this course include:

- Abstraction: Creating functions and classes
- Encapsulation and Information Hiding: Providing the correct interface
- Inheritance: Extending code developed in other objects
- Dynamic Binding: Polymorphism

13.1 Abstraction

Per Sebesta [1], abstraction is, "a view or representation of an entity that includes only the most significant attributes."

What are the minimum features or characteristics a thing must possess to be named that thing? What are the minimum features in a car? A plane? A writing instrument? With abstraction, we try to distill a concept to its most essential components. For example, in order to be considered an omelette, what characteristics must the thing have?

Per Dictionary.com, it is ”a dish of beaten eggs cooked in a frying pan until firm, often with a filling added while cooking, and usually served folded over.”

This definition makes no mention of the specifics of the filling, its texture, or the number of eggs used. Were one constructing a menu of all the restaurant's available omelette options, however, one would not *need* these data. Simply knowing the item *is* an omelette suffices. Ham and Cheese, Italian, Egg-white only – the specifics of the variations play no role in determining their rightful place on the menu in the omelette section.

By reducing an object to its most essential components, we simplify the resultant software. If every omelette definition included a place-holder for the number of olives present, this field would waste space in the majority of omelette recipes. What about the variation of green or black olives? Instead of including these fields in the definition of *every* omelette, these features only belong in the specific variety of omelette that includes them. Omelettes also possess a temperature, for one would not likely enjoy the dish after it sits on a plate for three hours. Should the omelette abstraction include its temperature?

The fewer hooks and connections an object offers, the less complex it is to use the object. Should the class include the instance's current temperature? If these data fields are used for a menu generator, these data possess no significance, so including them in the abstraction serves little purpose and requires additional space.

Note: Make certain the abstraction represents a class and not simply the role it plays in the code. That is, does the software warrant Mother and Father classes, or are these roles of a Person class with two instances of mother and father?

13.1.1 Process Abstraction

In Java, when one wishes to send output to the terminal screen, one typically uses some variant of `System.out.println("Driving on nine")`. Alternatively, one could adjust the individual pixel values for the appropriate display positions on a pixel-by-pixel basis to achieve this result. Doing so, however, proves unnecessarily tedious. The process of writing output to the screen is effectively the same when printing ”you could be a shadow” or ”beneath a streetlight.” Only the output values – the specifics of what one is printing at that time – that changes.

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

void _init();

int main(){
       _init();
}

void _init(){
       // ... insert initialization sequence here
       cout << "Initialized." << endl;
}
```

Figure 13.1: One example of process abstraction in the C++ language. The `_init` function abstracts away the concept of initialization.

Here, we see the need for *process abstraction*. Rather than setting the values for each pixel by adjusting the output value of a CPU register, one can group the essential characteristics of this concept into a function and then call *this* function by name. The `System.out.println` method does one thing, and it does so cleanly.

The C language supports the creation of functions, like Java, Python, C++, and Elixir, so all these languages support process abstraction. This does not make these all object-oriented languages, however, for process abstraction is only one feature in an object-oriented language.

13.1.2 Data Abstraction

In the C language, a **`struct`** represents a *record* of data. One may view a **`struct`** as the first, sputtering start toward data abstraction. Recall that with these data types one may begin grouping together like items into a single, monolithic entity. Rather than storing an array of student identification numbers and an array of grades, one may create a student abstraction through a structure, and inside this abstraction it includes both the grade and identification number, for these fields are closely related.

An album possesses a list of the names of contributing artists, and it is likely that when one inspects the bassist of a band that one would also want to parse the lead guitarist or drummer. Thus, keeping these fields collected together makes sense. Due to the *principle of locality*, which captures the

```
#include <iostream>
#include <vector>

int main()
{
    // init the object holding the specified values
    std::vector<int> v = {3, 20, 4, 40, 4};

    // Manipulate the object by adding to its end
    v.push_back(96);

    // Iterate and print values of vector
    for(int i : v) {
        std::cout << i << endl;
    }

    v.clear();
}
```

Figure 13.2: An example of an abstract data type in C++ using a `vector`. This object holds the data and includes methods to manipulate the items inside.

idea that computer processors tend to access the same set, or near adjacent set, of memory locations. This concept includes both the physical location in memory, but it also extends into the time domain. That is, after using a variable, the computer will likely use it again in a short period of time.

Although a step in toward object-oriented, these concepts fail to fully meet the object-oriented concept of data abstraction. Object-oriented languages support *abstract data types* (ADT). An ADT includes the data fields, like those in a record, but it also includes the functions, sub-programs, or methods necessary to manipulate the data. These objects group together both the items one wishes to hold, but also the functions necessary for adding or removing from the structure.

Abstractions within an object-oriented programming model communicate by passing messages. Rather than directly modifying or accessing a variable, one uses methods to reach into the class and get these data. It is rude for a waiter to reach into a customer's bag for payment – one must request money instead of directly taking it from someone.

Heuristic: A class should capture one, and only one, key abstraction.

13.2 Encapsulation and Information Hiding

The concepts of *encapsulation* and *information hiding* remain closely linked, but they serve different purposes.

Encapsulation is a strategy for achieving information hiding. [13]

Heuristic: All data should be hidden within its class, for if all data is public, how do we separate the essential? Where are the dependencies?

Heuristic: Beware of classes with many getters and setters, for they reflect poor design encapsulation.

13.2.1 Access Modifiers and Specifiers

Supporting the object oriented concepts of encapsulation and information hiding, programming languages like Java and C++ include special keywords to indicate the visibility of a given variable within a scope. The goal remains simplifying the interface to other components so that only the essential is exposed.

The C programming language provides no mechanism for access modification, for it does not include classes and the structures remain data focused. That is, structures in C hold data, but they do not include methods or the notion of operator overloading. Python fully supports classes, but it does not provide a method for access modification.

To indicate a variable or method is internal in python, one simply prefaces its name with an underscore character. This naming convention helps developers achieve much of the same encapsulation without forcing it in software.

In Java Access modifiers are the keywords which are used with classes, variables, methods and constructors to control their level of access. Java has four access modifiers: `public`, `private`, `protected`, and `package` (or *default*).

`package`	When no access modifier is specified, its visibility is limited within the package.
`public`	It has visibility everywhere.
`private`	It has visibility only within its own class.
`protected`	It has scope within all inheriting classes and objects in the same package.

Figure 13.3: Access specifiers in Java.

`public`	It has visibility everywhere.
`private`	It has visibility only within its own class.
`protected`	It has scope within all inheriting classes.

Figure 13.4: Access specifiers in C++.

In C++ In classes and structures, three keywords impact a variable's visibility within the current scope: `public`, `private`, and `protected`.

Friendly Modifiers

A friend function is a function outside the class with special permission to access the internal state of the class. Because it may access the internal workings of a class, the target class must specify that the friend function has access with the `friend` keyword before use. That is, a class must explicitly identify all external functions it considers friendly.

13.2.2 Naming Encapsulations

As software systems pass beyond the toy, educational level application and into real-world applications, they grow increasingly complex. Moreover, they likely include code from a number of sources. Some of these sources might be other, in-house developed applications, but they may also come from third-party sources. One might wish to include computer vision software modules developed by industry experts rather than developing one from scratch [1].

As these other tools become part of a software package, the global functions and variables they define *also* become part of the current software design. To help prevent these clashes, or *name collisions*, many languages include the concept of a *namespace*.

[1]hint, hint

```
// classes_as_friends1.cpp
// compile with: /c
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
   int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; } // OK
```

Figure 13.5: Friendly example from Microsoft.

```
namespace Fleet
{
    // a class object in the namespace
   class SuperDimensionalFortress
   {
   public:
       void Transform(){}
       bool isSelfAware(){}
   }

    // This is C++, so a function without a class in the namespace
    void CheckStatus(SuperDimensionalFortress){}
};
```

Figure 13.6: An example of namespace in C++

```
Fleet::SuperDimensionalFortress sdf;
sdf.Transform();
Fleet::CheckStatus(sdf);
```

Figure 13.7: Working with the fully-qualified name.

The use of a namespace helps developers organize their code into meaningful units. All the items within the same namespace may ”see” one another without a fully-qualified name. A namespace is a way to identify which variables or functions should take preference over others when they have the same name, and it allows developers to write code using shorthand – for the namespace means one need not include the entire qualification.

Namespaces in C++

If one does not include a namespace identifier in a source unit, then the compiler automatically places it in the *global namespace*. Keeping items in this namespace serves useful purposes in an educational environment, but in any meaningful program good form dictates that developers place their classes in an appropriate namespace. C++ places some of the most universally useful tools in the `std` namespace. These subroutines and objects create the *C++ Standard Library*.

In addition to defining a namespace at the global level, one might wish to break sub-sections of code into their own, distinct namespaces. That is,

```
using Fleet::SuperDimensionalFortress;
SuperDimensionalFortress sdf;
```

Figure 13.8: Bringing one identifier into *scope* with the `using` directive.

```
using namespace Fleet;
SuperDimensionalFortress sdf;
sdf.Transform();
CheckStatus(sdf);
```

Figure 13.9: Grabbing everything in the namespace and bringing it into scope.

one may wish to *nest* namespaces. Entities within a nested namespace have access to all other entities within the same, nested-namespace, but they may also access entities within their parent namespace.

On the other hand, entities in the outer namespace may not access the members of the inner namespace without qualification.

Namespaces in Java

In Java, one may group together related classes in a common `package`. Were one developing custom data structures, for example, one might elect to place them all in the package `datastructures`[2]. This way, when you reference your custom `Vector` object, it does not conflict with a similar structure in the Java Collections Framework.

All entities within the same package in Java have access to the `public`, `protected`, and unspecified access permission level (package-scope) of all other types within the package. That is, everything in the same package may peer into all but the inner-most, `private` workings of the objects.

13.3 Inheritance

Sometimes, one needs to build a program up from the ground by its bootstraps, but other times it's nice to suddenly gain powers due to your parent's status. The inheritance or an estate goes to the next-of-kin in a simple model

[2]Or `edu.sdsu.cs.datastructures` if you have an awesome 310 instructor

```
import java.util.ArrayList;
import java.util.List;

...

List<Integer> myList = new ArrayList();
```

Figure 13.10: Pulling in subroutines defined in other modules in Java through the `import` statement.

[3]. Object-oriented languages support this abstraction in some way.

Using inheritance, the new types defined by the software become a specialized *type* of the parent class. They possess an *IS-A* relationship. Using inheritance, we may establish a model for a simple `Duck` class. In this model, we want to simulate multiple duck types: Wood Ducks, Mallards, and Canvasback. All three of these animals possess the abilities to quack and fly.

For ducks, apparently, each type produces a slightly different call. Consequently, in this application, we want to specify that all ducks possess this ability, but we want to require every duck type to implement their own duck sound. For our purposes, the birds in this model all fly the same way, so they do not need unique behavior for that method.

Our goal is to:

- Create a simple, universal `fly` method all ducks can share.
- Require each implementing type to define their own quack noises.

We could implement this functionality directly in each specific class. Instead, we will add a level of abstraction to the problem. Although we want to actually model three specific duck types, we will begin by introducing a higher-level abstraction: the `Duck`. We do not intend for implementations to actually instantiate bland duck objects, but every mallard and wood duck share functions, and one might combine these functions into a parent class object so one only needs to write the code in one place.

One major advantage of Inheritance emerges in its ability to introduce new functionality into the inheriting classes without changing their source code. So far, we have discussed inheriting functions in advance and by design, but we may also use inheritance during the maintenance cycle. If one were to modify the `Duck` parent class by adding new functions and their

[3]Genetic testing may complicate this in the future.

```
class name : access_mode parent_class_name
{
  // Define the subclass here . . .
};
```

Figure 13.11: Basic structure for inheritance in C++.

associated code, then every existing class that inherits from Duck also gains that ability.

- Pro: Model relationships between objects
- Pro: Code-reuse
- Pro: Add new features
- Con: Introduces dependencies which may be difficult to unravel
- Con: Tight coupling

13.3.1 Inheritance in C++

We may model this problem in C++ directly using the built-in classes and the language's inheritance procedure. We simply need to define the base behavior in the Duck class, and then all children gain the ability.

To help support the concepts of information hiding, C++ permits access specifiers which modify the visibility of the inheritance. That is, one might wish to inherit the public members of a class in a way that hides those features from the outside world, for if you derive from a class with public functions, then those functions become a part of the child as well. This might clutter up the interface.

Consequently, C++ permits the developer to specify the way to treat the *public* methods and fields it receives from its base class. The modifiers follow standard C++ convention with: public, protected, and private. What use could this be – Implement a stack.

Virtual Functions

> The member function area has been declared as virtual in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived

```
#include <iostream>

namespace Targets
{
       class Duck
       {
       public:
           virtual void quack(void){};
           void fly(){std::cout << "Flap Flap" << std::endl;};
       };

       class WoodDuck : public Duck
       {
       public:
           void quack(void);
       };

       class Mallard : public Duck
       {
       public:
           void quack(void);
       };

       class Canvasback : public Duck
       {
       public:
           void quack(void);
           void fly(void) { std::cout << "Canvas Flap" <<
               std::endl;};
       };
};
```

Figure 13.12: An example of Inheritance in C++ using ducks.

```
using namespace Targets;

void WoodDuck::quack(void){
  std::cout << "Woody Quack!" << std::endl;
};

void Mallard::quack(void){
  std::cout << "Malignant Quack!" << std::endl;
};

void Canvasback::quack(void){
  std::cout << "Burlap Quack!" << std::endl;
}

int main(){

  Targets::WoodDuck fred;
  fred.quack();
  fred.fly();

  Mallard mary;
  mary.quack();
  mary.fly();

  Canvasback carl;
  carl.quack();
  carl.fly();

  std::cout << "Complete!" << std::endl;

}
```

Figure 13.13: Using the earlier duck example defining class methods outside their definition.

```
Woody Quack!
Flap Flap
Malignant Quack!
Flap Flap
Burlap Quack!
Canvas Flap
Complete!
```

Figure 13.14: Output produced after running the duck example.

> classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.

Declare the methods in the base class that software can redefine in derived classes.

Virtual functions act as a type of *interface* to methods in derived classes. All derived classes which provide implementations for these methods are said to *override* them. Although listed under inheritance, virtual functions help abstraction, encapsulation, and dynamic binding.

Keyword	Description
`virtual`	The method *may* be overridden
`=0`	The method **must** be virtual and it **must** be overridden
`override`	The method is meant to override a virtual method in the base class
`final`	The method is not meant to be overridden

Pure Virtual Base Classes

One may specify individual methods are virtual and meant to be overridden, but what if the class definition does not include any member fields, so it does not need state? This class might not need a Constructor. What if every method defined were virtual and the file does not provide implementations. That is, what if the class definition file only required implementing classes to perform work? The Class itself does nothing except specify the interface one plans to use when interacting with objects of this type.

```
class Polygon {
  protected:
    int num_sides;

  public:
    virtual void area() =0;
}
```

Figure 13.15: An example of an abstract class. No meaningful default exists for the `area` method. By declaring it `virtual` we require all children classes to provide the code for this message.

Abstract Base Classes

Sometimes, one wishes to provide a class implementation with *most* of the software provided so that developers only need to spend time working on creating only the most relevant code. This simplifies work by allowing significant code-reuse. One might wish to provide a default implementation and require modified and new implementations to then override this behavior.

What if no meaningful default exists?

Multiple Inheritance

The C++ language includes *code* inheritance from multiple sources, but it did not do so in the original C++ language definition, and the language committee only included it in response to an apparent user demand. The demand, in part, was due to software developers at the time not understanding what they requested. By introducing the feature, the language created a maintenance apocalypse. Liberated from the constraints of good software design, developers were finally able to create garbled, intermingled classes without a clear inheritance chain!

> Heuristic: If you have Multiple Inheritance in your design, assume you made a mistake and prove otherwise.

Done Wrong Imagine trying to model an automobile through multiple inheritance. We can say that our Car object inherits from the Cockpit, Drive-train, and Engine. The Cockpit includes a Radio and Steering Wheel, and the Radio includes Electronic Components and Screws.

You can turn a Screw, and you can turn a Steering Wheel, and you can turn the knob on the radio. Because with multiple inheritance the inheriting class gains all the capabilities of its parent, in this example, everything you can do with a screw, you can do with a Car, and this does not make sense. Certainly, although a car operator CAN turn screws in the vehicle, this is not something typically performed while driving.

13.3.2 Inheritance in Java

In response to the maintenance nightmare multiple-inheritance unleashes on a code-base when used inappropriately, Java took the position that a class may only directly inherit *code* from one class. Thus, two classes cannot provide a child class with instance variables.

Objects rarely fall into a single abstraction, however, so this restriction places severe limits on the utility of Dynamic Binding and impairs software development. Consequently, Java allows classes to inherit *requirements* from any number of classes. These special parent classes serve as *interfaces*. They provide absolutely no code to inherit – instead, they possess method definitions.

Java uses an interface file to define an abstraction without providing any implementation details. Pure virtual classes in C++ serve a similar purpose. Although these other classes provide no code, any implementing class may then, through type inheritance, act like an instance of the parent class.

You may only *extend* the code of one class in Java, but one my *implement* any number of additional requirements.

Note: Java 8 introduced the `default` keyword in interface files, and this now allows developers to provide interface files *with* code that classes can inherit from. Consequently, Java 8 now supports multiple inheritance.

13.3.3 Inheritance in Python

Python includes support for inheritance in its class design. In fact, every class in Python inherits from the super-class `Object`. This common class establishes the framework for the class itself.

```
abstract class IteratorHelper<E> implements Iterator<E>{
   protected int index;
   protected Node<Entry<K,V>> [] nodeArray;

   public IteratorHelper(){
      // setup the same for everyone
   }
   public boolean hasNext(){
      return index < size();
   }

   // this is the one abstract method every child must implement
   public abstract E next();

   public void remove(){
      throw new UnsupportedOperationException();
   }
}

class KeyIteratorHelper<K> extends IteratorHelper<K>{
   public K next(){
     if(!hasNext()) throw new NoSuchElementException();
     return (K) nodeArray[index++].getKey();
   }
}

class ValueIteratorHelper<V> extends IteratorHelper<V>{
   public V next(){
      if(!hasNext())throw new NoSuchElementException();
     return (V) nodeArray[index++].getValue();
   }
}
```

Figure 13.16: Useful inheritance in Java for an Key and Value iterator

```
// A simple Java interface file illustrating default methods
interface ExampleInterface
{
    public int countThings();

    // default method
    default void status()
    {
      System.out.println("Having this makes you feel good.");
    }
}
```

Figure 13.17: Java 8 introduces **default** methods in interface files.

```
class DerivedClassName(BaseClassName):
    pass
```

Figure 13.18: The basic template for declaring a derived class in Python.

Method or Attribuite	Use
`self`	Similar to `this` in Java or a pointer to the base class in C++
`__init__`	Constructor
`__str__`	The `toString` method or `<<` C++ operator
`__dir__`	Returns a list of attributes supported in the class
`__hash__`	A unique integer. Two equivalent objects must produce the same code.
`__doc__`	Provides the class' docstring.

Multiple Inheritance in Python

Unlike Java, Python supports Multiple Inheritance. Because all classes in Python inherit from Object, every class with multiple inheritance *must* include an inheritance diamond pointing to the common Object class. To resolve conflicts in naming, Python attempts to bind methods by searching *almost* linearly, depth-first through the inheriting classes. Python cannot reliably guarantee it always fulfills the call from left to right due to limits

```
class Person:
   def __init__(self, first, last):
      self.firstname = first
      self.lastname = last

   def Name(self):
      return self.firstname + " " + self.lastname

class Wizard(Person):

   def __init__(self, first, last, mana):
      Person.__init__(self,first, last)
      self.mana = mana

   def GetMana(self):
      return self.Name() + ", " + self.mana

settler = Person("Preston", "Garvey")
quest_giver = Wizard("Morgan", "la Fay", "1000")
```

Figure 13.19: A basic example of class inheritance using Python.

```
class DerivedClassName(BaseClassName, OtherClassName,
    YetAnotherClassName):
   pass
```

Figure 13.20: An example of multiple-inheritance in Python.

imposed by the software's design (e.g., calls to `super` in the methods might interfere with this).

13.4 Dynamic Binding or Dynamic Dispatch

Frequently when coding software tasks, one must repetitively perform the same operation on collections of dramatically different objects. The single operation they share may be their only commonality. That is, the only thing the two objects have in common may be the desire to print their description or ability to periodically update themselves in response to a changing clock. Let us create an example of a Non-Player-Character and it's current mood.

We want to model an entity with a dynamically adjusting description based upon the character's current mood. When one looks at the character, one should receive some type of useful reflection of their mood. We will use the general format, ”The character looks <MOOD HERE >.” Implementing this by storing an integer we evaluate to compute the mood, where negative scores reflect poor moods and positive scores represent good moods could quickly establish a working system, but what if we want more? A software update might want to establish a more advanced mood evaluation strategy. How do quests impact the character's mood? Its interactions with other NPCs?

Some characters, like simple vendors, might use something simple, but more interesting characters – characters central to the plot – warrant more complicated mood calculations. Should we create two classes, one for primitive characters and one for advanced characters each descending from some common character? Does anything *else* change between these characters, or is it just the intelligence? What if we created a single `NonPlayerCharacter` class which *contains*, (i.e. a HAS-A relationship), an internal `Mood` object.

Moreover, if we declare `Mood` as an *interface* or *pure-virtual* class, then we could create child classes, which type-inherit `Mood` but calculate their current mood in dramatically different manners. The simple vendor or NPC extra in a crowd could use a trivial mood calculator – perhaps one even hard-coded inside – and advanced characters could use computationally intensive algorithms. In the code, the developers do not need to know the specifics of the individual `Mood` child classes. They only need to know that every child of `Mood` contains a method that returns a string and describes the current mood.

The class may then communicate its **message** to the mood object through the `getMood` method. One could even change a NPC's mood calculation object dynamically on the fly at run-time. Perhaps a spell briefly changes how a NPC feels about the player, one might wish to swap in a new `Mood` object. Dynamic dispatch makes these type of run-time changes possible. Using a language with only **static binding**, one could only find similarities at compile time.

One may only determine the type of a polymorphic variable or method at run-time.

Chapter 14

Design Patterns

Using the features provided by an object-oriented language, one may begin creating greater abstractions. For example, rather than saying, ”Use polymorphism to create a method in every object called `next`, and then use this to retrieve the next item in a sequence,” we say something like, ”make that object Iterable.” Design patterns combine encapsulation, abstraction, and dynamic binding to help better structure code [14].

Although not a course in object-oriented programming, no discussion of the topic feels complete without a few meaningful examples. Design patterns break down into three major categories.

- Behavioral: Identify common message patterns between objects and create abstractions around those.
- Creational: Instantiating an object in a specific way.
- Structural: Organizing different objects to build new functionality or frameworks

14.1 Behavioral: Iterator

Modern programming languages support the *for-each* conditional statement. The idea being that instead of iterating from a zero index up to the index of the stopping point, rephrase this concept and write the software to work with the items. For each *item* in this collection, do the following. The `Iterator` object provides this mechanism. It manages sequencing through a collection of objects. Typically, one pairs this with the `Iterable` pattern.

```
int main()
{
   int arr[] = { 1, 2, 3, 4 };

   for (int curItem : arr)
      cout << curItem << endl;
}
```

Figure 14.1: A C++ program demonstrating iteration using a *for-each* statement.

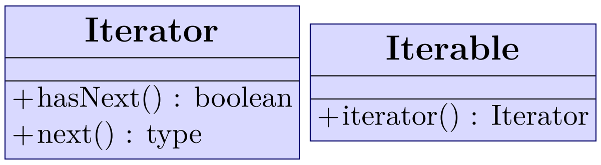
```
#include <iostream>
#include <vector>

using namespace std;
int main()
{
   // Iterating over whole array
   vector<int> v = {0, 1, 2, 3, 4, 5};
   for (auto thing : v)
      cout << i << ' ';
}
```

Figure 14.2: A C++ program demonstrating its *range-based* iteration.

An iterator provides the interface between algorithms and the data structures, or containers, that contain them.

`Iterable`	This object is able to produce an `Iterator`
`Iterator`	This object handles an iteration



14.2 Behavioral: Comparator and Comparable

How does one need to sort a collection of items from first to last? Given numbers, or letters, this seems trivial. For example, if you were given: 19, 22, 14, 4, and 11 and then told to place them in ascending order, you might return: 4, 11, 14, 19, and 22. If one were placing items into their *natural order*, this would be valid. What if we wanted them in inverse order? What if they weren't numbers at all?

The original sequence (19, 22, 14, 4, and 11) – are already in increasing order, for they represent the driver finishing order for the Saturday, April 13th Richmond NASCAR race. Unfortunately, Joey Logano finished second.

14.3 Structural: Adapter

What happens when the vendor who supplied the GPS module your company used for the past five years stops developing your module or changes the command protocol? If your software makes direct calls to the third-party tool directly, then every class that touches it must change. Consequently, swapping out the component becomes expensive. Instead, what if the developers coded their main application using a custom, internal component?

Rather than making calls directly to the soon-to-be-obsolete part, create software that sends messages to an object with an interface under your control. Then, if the vendor changes, one only needs to update the `Adapter` class. This limits the changes to a single class. One may also use an adapter to convert an existing data structure to a new type.

14.4 Creational: Object Pool

In situations where building an object requires extensive time, what if one simply instantiates a block of them once and then manages their distribution? Rather than directly getting instances of a class, request these from the `Object Pool`. When one finishes using the object, one simply returns it to the pool. Obviously, objects in the pool need some method to clear or reset themselves, but otherwise the allocation completes only once – when they are loaded into the pool.

Particle systems and objects in game worlds frequently use object pools, for the sparkles radiating from a laser always look the same; Vehicles simply need their damage reset.

Chapter 15

Memory Management and Error Handling

15.1 Garbage Collection

Recall when one allocates memory on the heap in C++, one must manually return these or they remain in place for the life of the program. Should developers forget to release these before program exit, the operating system will *almost* certainly free up the heap memory allocated for that process. For persistent, real-time systems, however, the process may never terminate, so memory the application fails to return is wasted.

Some programming languages include a mechanism for handling these loose-ends. Rather than require the developer to explicitly instruct the machine to return the memory, implement a sub-system that runs behind the scenes and automatically detects when memory is unused.

Garbage collection seeks to efficiently manage dynamically allocated memory independent of the source unit. Both Java and C# supply garbage collectors. Python automatically allocates and deallocates memory as needed, so it includes garbage collection by default, but it does so slightly differently.

Why would one chose a language without garbage collection? In some real-time applications, the OS simply lacks the resources to support the action. Garbage collected languages certainly help with several targeted errors, but this does not eliminate the error, and it requires additional overhead, for one cannot trust the Java Virtual Machine to eliminate unused references on systems without JVM running.

Garbage collection requires computation, so it must receive CPU time. When it receives this time remains up to the implementation. It might run

```
{
  int *dangling;
  {
    // declare something on the stack
    int onstack = 320;
    // point the outside variable at it
    dangling = &onstack;
  }
  // stack frame pops, but dangling points to an address inside!
}
```

Figure 15.1: A dangling-pointer bug in C++ code blocks.

```
{
  Robot *target = new Robot("IG-88");

  // some code
  target = new Robot("HK-47");
}
```

Figure 15.2: A memory-leak after pointing target to a new object. IG-88 is still floating around on the heap.

concurrently as a scheduled task, or it might trigger in response to certain actions.

15.1.1 Targeted Errors

Recall that a *dangling-pointer* develops when an aliased variable falls out of scope. That is, a pointer stores a reference to an invalid memory address.

Memory leaks also manifest in C and C++ when one loses a reference to dynamically-allocated heap memory. This might happen when switching pointers. This error tends to manifest subtly as programs grow more complicated and objects begin interacting with one another.

Another error that garbage collection seeks to remedy is the *double-free* bug [1]. It emerges when a program attempts to return to the heap memory already freed. This type of error may appear due to conditional logic and

[1] Possibly discovered at a Doubletree

```
GiantFightingRobot *target = new GiantFightingRobot("Mazinger");
Thing *one;
one = target;

// some code
if( one->isDestroyed() ) free(one);

// cleanup before exit
free(target);
```

Figure 15.3: A double-free bug in C++. The `GiantFightingRobot` descends from `Thing`.

good-intentions. That is, this error appears when one overly-aggressively frees memory. One should make sure cleanup happens only once for any variable in languages without garbage collection.

Garbage collection represents *one* solution to these errors. Alternatively, consistent programming practices and software engineering will help mitigate them. Readily-available tools in languages without a garbage collection scan either source code or profile a running program and effectively identify their presence in most of these languages, so by no means are they a plague in current programming.

15.1.2 Tracing

The garbage collectors provided by both the Java Virtual Machine and the .NET framework in C# use a strategy of reachability. The collector identifies all heap memory allocations it may *reach* by tracing a path from all current reachable objects. For this strategy to work, the garbage collector needs some base set of root objects it never collects. These could include the stack, program registers, or other system-wide objects. These would be the essential components of any program.

Then, stepping through each one, it identifies *every* location accessible through a root or something contained recursively therein. The collector considers anything referenced from a reachable object as reachable.

Java Garbage Collection and Security

As part of efficient memory management, the JVM periodically moves objects in memory. A system memory space might become peppered with small allocations which make allocating a contiguous block of memory impossible. Recall that the `malloc` and `calloc` operations in C *guarantee* the data will be together sequentially without gaps. Thus, the largest block of memory the system may provide is defined by the biggest gap between objects in memory.

When Java moves these objects, it leaves behind the dirty memory location. Thus, even if an object takes care to keep private data in non-volatile memory and clear them before destruction, a full read of the system memory would reveal the original data potentially lingering in an unused cell. Several early attacks targeted the garbage collector for this purpose.

In much the same way snooping through a real trash-can can reveal credit-card information, so can scanning all the objects in the garbage collector as well. Moreover, because these copies occur behind the scenes, the developer cannot control when they happen. `https://www.cs.princeton.edu/~appel/papers/memerr.pdf`

Thankfully, the JVM garbage collector remains the focus of intense study, and these types of errors appear less frequently.

Weak References

Java includes several data structures which support weak references. Unlike the references used in tracing, a weak-reference does not prolong the life of an object like a typical pointer reference would. The collector considers *weak-references* as useful, but possibly invalid, data. Consequently, the main application can free the variable by removing its reference without having to notify or update the weak reference.

In the Observer design pattern, the publisher must maintain a link to every observer object. If implemented as *strong* references and stored in a typical aggregate data structure, their presence in the publisher will prevent them from being garbage collected.

A `WeakHashMap` prevents this behavior, for the garbage collector *will* collect objects if they only appear in the `WeakHashMap`.

15.1.3 Reference Counting

Python's garbage collector uses a reference count for each allocated block of memory. When the reference count reaches zero, the collector knows the area is unused, so it may return the area to the system for use.

An object with a *reference cycle* will not automatically disappear when garbage collection runs. This memory area, however, may very well be

```python
import gc
num_collected = gc.collect()

print("GC {} items.".format(num_collected))
```

Figure 15.4: Calling the garbage collector in Python for long-running server programs. This code should collect zero objects.

```python
import gc

def cycle():
  tmp = {}
  tmp[0] = tmp;

for count in range(320):
  cycle()

num_collected = gc.collect()
print("GC {} items.".format(num_collected))
```

Figure 15.5: The `cycle` function creates a `tmp` dictionary with a self-reference, so it will always have a reference to itself.

otherwise unreachable, for if the only reference to an object is through itself, the client program cannot access the data. Python's garbage collector will not automatically detect this, so the user must initiate garbage collection manually to recover the memory.

The automatic garbage collector uses reference counting, but manually triggering garbage collection will collect unreferenced objects by the main program. Thus, Python mixes reference counting which it performs automatically with manually-activated tracing.

General heuristics about when to trigger a manual garbage collection:

- Python stops automatic garbage collection when low on memory
- Consider running it after the program finishes loading, booting, and initializing.
- Prior to initiating time critical sections of code where background collection could impact performance.
- After blocks of code that free large blocks of variables (perhaps when reading in data)

15.1.4 Destructors and Constructors

In Java, the JVM calls the `finalize()` method attached to every object. For almost all applications, the default method provided in the Object class remains sufficient, but for objects with complicated resource allocations, users may wish to override this behavior. One might wish to disconnect from an external server, shutdown resources, or notify an Observer. To do so, one need only override the method in the class definition.

The C++ language also provides a default destructor, and for many applications the default suffices. If an object allocates heap memory, however, one must explicitly remove the reference and free the memory. Failing to do so might lead to a memory leak. Destructors are noted by a tilde appearing in front of the class name.

15.2 Exception Handling

Exception: a person or thing that is excluded from a general statement or does not follow a rule.

– Dictionary.com

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

class Label{
private:
  int serno;
public:
  Label(int number){serno = number;}
};

class Orange{
private:
  Label *farmer;
public:
  Orange(){farmer = new Label(1);}
  ~Orange(){free(farmer);}

};
```

Figure 15.6: Example of a class with a destructor and constructor in C++. The allocation of Heap memory requires a `free` to avoid memory leaks.

Typically, failing to catch an exception results in a catastrophic failure. At a minimum, it clunks the program flow and slows down performance. Exceptions handle situations outside standard program flow. They might cover errors like the inability to get more memory, open a file, or reflect invalid input. They also cover invalid parameters provided to objects.

Should a constructor throw an exception? What are the implications?

15.2.1 Java: try-catch

https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/

Java requires the try-catch mechanism throw a Object of type `Throwable`. One cannot simply throw numbers or characters as a result. Java allows one to catch the general `Exception` object which emulates the C++ `catch(...)` behavior. Java also allows one to include the `throws` keyword to indicate an object produces a particular error under knowable state.

Checked Exceptions are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword. In general, these are errors one might easily recover from with some other valid input or another try.

> The price of checked exceptions is an Open/Closed Principle violation. If you throw a checked exception from a method in your code and the catch is three levels above, you must declare that exception in the signature of each method between you and the catch. This means that a change at a low level of the software can force signature changes on many higher levels. –Robert C. Martin, Clean Code, page 107

The open/closed principle states that software modules should be open to extension and closed to modification.

> When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with"bad" design. [...] The open-closed principle attacks this in a very straightforward way. It says that you should design modules that never change. –Robert C. Martin, The Open-Closed Principle, 1996

> Use checked exceptions for conditions from which the caller can reasonably be expected to recover. [...] By confronting the API user with a checked exception, the API designer presents a mandate to recover from the condition. –Joshua Bloch, Effective Java, page 244

Unchecked Exceptions In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.

Java includes the `finally` block which *always* executes after the try-catch block completes. One might use this to close external resources. No matter what happens in the try-catch behavior, it will always call finally last.

> A well-documented list of the unchecked exceptions that a method can throw effectively describes the preconditions for its successful execution. –Joshua Bloch, Effective Java, page 252

```
class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("./out.txt");
        BufferedReader fileInput = new BufferedReader(file);

        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Figure 15.7: This Java code will not compile because it does not catch `FileNotFoundException`.

15.2.2 C++: catchall

All C++ compilers support the catchall format, but some consider it dated and bad form like using raw pointers. Instead, the C++11 standard library includes a new exception type to work with.

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

Instead, C++11 introduces the concept of an `std::current_exception` object. This object, captures the current exception object and creates an `std::exception_ptr` strongly holding a copy to the exception object. Creating this object does not, itself, cause a new exception, so one must typically:

1. Catch the current exception in a `catchall`
2. Build a `current_exception` object from the try-catch context
3. Rethrow the exception as a `current_exception`
4. Catch the std::exception_ptr thrown and continue error analysis

```
int main()
{
   int x = 320;
   char *ptr;

   try {
      if( x < 0 )
      {
         throw x;
      }
      if(ptr == NULL)
      {
         throw "Dead stars still burn.";
      }
   }
   catch( int x )
   {
      std::cout << "Suboptimal: " << x << std::endl;
   }
   catch (...)
   {
      std::cout << "Exception occurred: exiting "<< std::endl;
      exit(0);
   }
}
```

Figure 15.8: Example of the try-catch mechanic in C++. Note the use of a catch-all block.

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr)
{
   try {
      if (eptr) {
         // Rethrow the excpetion using an eptr object so we
            process it here
         std::rethrow_exception(eptr);
      }
   } catch(const std::exception& e) {
      std::cout << "Caught " << e.what() << "\n";
   }
}

int main()
{
   std::exception_ptr eptr;
   try {
      // ...
   } catch(...) {
      eptr = std::current_exception();
   }
   handle_eptr(eptr);
}
```

Figure 15.9: The updated exception handling procedure in C++11 using the standard library. This method provides an exception object to work with. [2]

Chapter 16

Generic Programming

Generic programming facilitates designing algorithms and data structures in terms of their operation and use and not the specifics of what they hold. It attempts to abstract out the key concepts by deferring many of the implementation specifics until the point of instantiation. A queue for coffee, for example, provides the same operations as a queue for cattle on a dairy. The items in line, be they coffee or people, sequentially exit in the order they arrived. Thus, one can program all the operations on the queue independent of what goes inside.

Languages provide differing levels of support for Generic programming, and they tend to refer to it in slightly different terms. Be it *generics* in Java and Python or *templates* in C++, generic programming ultimately seeks to reduce the amount of code developers need to write and maintain.

The Python programming does not require any formal syntax with generics. Instead, it uses a strategy called *duck typing*. The idea follows the saying, ”if it looks like a duck and quacks like a duck, it's a duck.” Instead of trying to verify the operation is permitted at compile time, Python assumes it is and acts accordingly.

16.1 Java Generics

Initial versions of the Java language lacked type-safe, generic support. Rather than include extra syntax, like C++ does when it identifies a generic function or class, it simply relied upon the inheritance tree to achieve this functionality. Recall that technical authors frequently employ the use of 'recall' to make their sentences seem more formal. Also, all objects in java descend from the Object super-class.

Rather than creating a linked-list for `Droid` objects, the early Java programmer envisioned it as a linked-list of type `Object` which the developer understood to be `Droid`. That is, the compiler possessed no way to verify or enforce that only `Droid` objects entered the list. Because everything descends from `Object`, a linked list in this manner might contain a mixed bag of `Droid`, `Duck`, and `Exception`.

With the Java 1.5 update, the language began including additional syntax for generic support. This syntax permits compile-time type checking to verify the software uses an object in the intended fashion. The extra notation helped catch errors as well as supplying additional documentation to the reader about a class' intended use.

Java's updated generic implementation, however, serves only as syntactic sugar over the original method. Behind the syntax, the Java compiler converts the generic references indicated in the brackets into casts it then applies to the `Object`. That is, verifies the objects inside the class match the appropriate type at compile-time, but it then converts the code into a new object which uses casts inside this single object.

The process of *type-erasure* removes the generic types identified in the generic object and converts it into a single, non-generic object capable of containing any style `Object` with casts to each of the generic types used. Consequently, an `ArrayList<Integer>` has no way to tell itself apart from a `ArrayList<Float>`, for they both become the `ArrayList` with the appropriate casts inside.

This feature serves as the root of data structure developers familiar warning: `Unchecked cast from Object to Integer`. Java warns the developer that the item referenced may, very well, **not** be the expected type.

Because there is only one copy of a static class at runtime, all type-specifications of a generic object use the *same* static variables. A static counter in `MyCounter<Integer>` references the exact same variable as one in `MyCounter<Long>`.

One may *not* call a generic constructor because, due to type-erasure, at run-time the program possesses no way to know what to instantiate, for it has broken free from the chains of typing. It follows that one may not perform a `new T()` operation if the program cannot distinguish `T` between a `Droid` or a `ArrayList`.

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0);
```

Figure 16.1: An example of generic syntax motivation in Java. Generating a compile time error here catches an error before run-time, but this framework lacks the capability. Provided by Wikipedia.

```
List<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0);
```

Figure 16.2: The updated generic syntax catches these types of type error at compile-time. Provided by Wikipedia.

16.2 Templates in C++

Templates support generic programming and design abstraction. They represent a general method for creating source code to accomplish a task. In C++, the process of generating the source code from the template is called *template instantiation*. The resulting template code is a template *specialization*. Multiple specializations may result from a single template.

One cannot explicitly state a template's requirements in code, and the user must glean this from the design. It behaves somewhat like a macro substitution. One can easily create a syntax error by calling a macro with invalid input. Likewise, one might introduce an error by instantiating a template in C++ with an object that doesn't support the required types or includes nonsensical values.

Templates in C++ hide the intended abstraction from the compiler whereas those in Java help enforce it.

Class Templates provide a way to specify that some of the class' members come from templates.

Function Templates identify that a function, independent of a class, accepts generic parameters.

```
template <class Receiver>
class SimpleCommand : public Command {
  Action _act;
  Receiver *_rec;

public:
  typedef void (Receiver::* Action)();

  SimpleCommand(Receiver* r, Action a) :
    _rec(r), _act(a) {}

  virtual void Execute(){_act->Execute();}
}
```

```
m1a1 = Tank();
SimpleCommand<Tank> tnkOff = SimpleCommand(m1a1, &Tank::powerOff())
tnkOff.Execute();
```

Figure 16.3: An example of a C++ template for the Command design pattern in *Design Patterns*.

Non-type Parameters extend beyond simply specifying that a type in the code is generic, one may even supply generic values. This functionality extends C++ templates into a new level of potential complexity.

```
template <class T, int N>
void DynamicArray<T,N>::set(int x, T value) {
  storage[x]=value;
}
```

16.2.1 Metaprogramming with Templates

http://people.cs.uchicago.edu/ jacobm/pubs/templates.html

```
template <int n>
class Fact {
public:
  static const int val = Fact<n-1>::val * n;
};

class Fact<0> { public: static const int val = 1; };
```

Chapter 17

Functional Programming

Functional programming gravitates toward several central themes:

- Immutability of Data: Errors emerge when the data an arbitrary algorithm uses changes in another thread.
- No Side Effects: The order of operations should not impact the results.
- Recursion: The use of function calls to perform iteration instead of loops.

Each of these address problems that frequently manifest in concurrent programming situations like those on the Internet or in multi-processor systems. Most modern programming languages support the idea of a *first-class function* – if not directly, then through a library addition. Java, for example, provides very little built-in support for functional programming, for one cannot use functions the same way as we do in C++ and Python. That is, one cannot set the value of a variable to a function.

> In computer science, a programming language is said to have first-class functions if it treats functions as first-class citizens. This means the language supports passing functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures.
>
> –Wikipedia

A *higher-order* function accepts a function as one of its input parameters or returns a function as a result. Functions in Python's map, filter, reduce algorithms frequently accept a function as the input parameter.

```
def recsum(x):
    if (x===1):
        return x
    else:
        return x + recsum(x-1)
```

Figure 17.1: A general recursive sum function

```
def tailrecsum(x, running_total=0):
    if (x===0) :
        return running_total
    else :
        return tailrecsum(x-1, running_total+x)
```

Figure 17.2: A tail recursive sum function

Pure functions produce output exclusively based upon its input parameters. That is, these functions use no stored state or external data to complete their operation. The mathematical operations provide an excellent illustration of a pure-function ideal. Addition remains independent of the temperature, barometric pressure, or previous winner.

Anonymous functions are not associated with a function label. These frequently take the form of *lambda* expressions. Alanzo Church proposed a Turing-complete, formal system for expressing computation based on functions.

Tail Call Recursion, as previously discussed, requires the creation of an activation record, and many recursive algorithms possess the potential for unbounded stack growth. That said, when the recursive call is the last item the function performs before exiting, the compiler may optimize the entire recursive function into an iterative while loop. This style of optimization remains essential in a functional compiler.

17.1 Python

As demonstrated previously with the *decorator* function, Python supports the functional style of programming. One may obviously create closures with the language, but it also supports *lambdas* and anonymous functions.

```python
def read_lines(filename):
    with open(filename, 'r') as the_file:
        return [line for line in the_file]

ex = read_lines("sample.txt")
```

Figure 17.3: A functional method for opening files in Python

```python
add_it = lambda a, b: a + b
add_it(1,1)
add_it(10,-20)
```

Figure 17.4: A basic lambda definition in Python

17.1.1 Map, Filter, and Reduce

Although Python supports functional programming and includes the map, filter, and reduce algorithms, *list comprehensions* better fit the coding style.

17.2 Functional in Java

Until Java version 1.8, the language provided very little functional programming support. Designed as an object-oriented language, many of the functional concepts appear antithetical to Java's syntax and design, for one cannot display text on the screen in Java without a class. That said, some functional concepts fit well with any programming style. Lambda expressions and anonymous functions are convenient syntax for complicated operations, so they found a place in the language itself.

Functional Interfaces in Java are interfaces which include a *single* method definition. Then, *lambda expressions* [1] may subsequently define the method to use as needed.

[1] https://www.geeksforgeeks.org/lambda-expressions-java-8/

```python
new_read = list(map(lambda a : a.upper(), read_lines("sample.txt")))
```

Figure 17.5: A map lambda expression in Python

```python
def my_filter(the_test, data):
    return Map(
      datum for datum
      in data if the_test(datum) is True)
```

Figure 17.6: The filter algorithm in Python

```python
import functools
import operator

the_data = [x ** 2 for x in range(10)]
print(functools.reduce(operator.add,the_data))
print(functools.reduce(operator.mul,the_data))
```

Figure 17.7: Reducing in Python.

```python
def test_func(datum):
  return (datum * 2) + 1

def my_comp(num):
  return [test_func(x) for x in range(num)]

def my_mapping(num):
  return list(map(test_func, range(num)))
```

Figure 17.8: List comprehension and a map applied to the same function.

```
interface FuncInterface
{
   void singleFunction(int x);
}

class Test
{
   public static void main(String args[])
   {
      FuncInterface fobj = (int x)->System.out.println(2*x);

      fobj.singleFunction(160);
   }
}
```

Figure 17.9: A basic Java Functional Interface.

17.3 Functional in C++11

Although not a functional programming language, C++11 and later include support for some popular functional features. With each subsequent revision, the language evolves to expand its functional capabilities. The `functional` header file includes several useful programming abstractions so one need not develop these basic ideas from scratch.

17.3.1 Lambdas, Anonymous Functions, and Closures

> A closure is an instance of a function, a value, whose non-local variables have been bound either to values or to storage locations.
>
> –Wikipedia

Sometimes, when working with higher-order functions, one needs a small, special-purpose function. When infrequently used, creating a named function might introduce more coding and complexity than necessary. In these situations, one might use an *anonymous function*. Anonymous functions in C++ take the general form: `[capture](parameters) -> return_type function_body`

```
[](int x, int y) -> int { return x + y; }
```

```
std::vector<int> some_list{ 1, 2, 3, 4, 5 };
int total = 0;
std::for_each(begin(some_list), end(some_list), [&total](int x) {
  total += x;
});
```

Figure 17.10: This computes the total of all elements in the list. The variable total is stored as a part of the lambda function's closure. Since it is a reference to the stack variable total, it can change its value.

[]	Nothing used in outer scope.
[*identifier*]	Copy *identifier* by value
[=*identifier*]	Copy *identifier* by value
[&*identifier*]	Copy *identifier* by reference
[&*first*,=*second*]	Copy *first* by reference *second* by value

Figure 17.11: Lambda capture specifiers if the object should be copied by value or reference.

One may *capture* variables in the greater program scope by including them in the square brackets.

Bibliography

[1] R. W. Sebesta, *Concepts of Programming Languages*. Pearson, 10th ed., 2012.

[2] J. Backus, “Can programming be liberated from the von neumann style?: A functional style and its algebra of programs,” *Commun. ACM*, vol. 21, pp. 613–641, Aug. 1978.

[3] P. J. Landin, “The Mechanical Evaluation of Expressions,” *The Computer Journal*, vol. 6, pp. 308–320, 01 1964.

[4] Google, *Google C++ Style Guide*. Google, Inc., 2019.

[5] Isocpp, “The c++ core guidelines.” `https://github.com/isocpp/CppCoreGuidelines`, May 2019.

[6] Epic, “Coding standard.” `https://docs.unrealengine.com/en-us/Programming/Development/CodingStandard`, 2019.

[7] “Pep-8 style guide for python code.” urlhttps://www.python.org/dev/peps/pep-0008/, 2019.

[8] W. Commons, “File:von neumann architecture.svg”’ `https://commons.wikimedia.org/w/index.php?title=File:Von_Neumann_Architecture.svg&oldid=262902966`, 2017. [Online; accessed 28-January-2019].

[9] Wikibooks, “Java programming/byte code — wikibooks, the free textbook project,” 2017. [Online; accessed 28-January-2019].

[10] B. W. Kernighan and D. M. Ritchie, *The C programming language*. Prentice-Hall Englewood Cliffs, N.J, 1978.

[11] Jraleman, “C programming language: Passing a function as a parameter,” Nov 2016.

[12] A. One, “Smashing the stack for fun and profit,” *Phrack*, vol. 7, November 1996.

[13] E. V. Berard, *Essays on Object-oriented Software Engineering (Vol. 1)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[14] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 ed., 1994.