Technische Universität München

Engineering Informatics I

A C Primer

Dr. Jonas Pfoh
Technische Universität München
Computer Science Department

Winter Semester 2013 - 2014

This document is meant as a basic C primer for someone with no programming experience. For this reason, I will try to keep the information as accessible as possible. This document *should not* be mistaken for a complete or thorough C reference. For such a reference, I suggest the following book:

The C Programming Language (2nd ed.)
Brian Kernighan & Dennis Ritchie

1. C Basics

This chapter will cover the most basic C syntax. We will cover the basic structure of a C program, variables, operators, and control structures.

1.1. Hello World

The following code listing shows a very basic C program.

```
1 #include <stdio.h>
2 
3 int main(int argc, char **argv){
4   printf("Hello World!\n");
5 
6   return 0;
7 }
```

Listing 1.1: A basic C program

Begin by copying the text in Listing 1.1 into a text file on the server (or any Linux environment) and calling that file *hello_world.c*. A very simple text editor that you can use in the beginning is *nano*. Once you have the code copied into the file and the file saved, you may compile it. The compiler we will use is *gcc* and is quite powerful. By default, *gcc* will call your program *a.out* if there were no errors. Simply calling a.out should now output "Hello World!" to the console. These steps and what they look like on the console are depicted in Listing 1.2.

```
$ nano hello_world.c
$ gcc hello_world.c
$ ./a.out
Hello World!
$
```

Listing 1.2: Compiling and running your first C program

Now that we have compiled and ran our first program, let's take a closer look at what it is doing. There is actually quite a bit happening here, we will break it down one-by-one. First, in Line 1 we see an `#include` preprocessor statement followed by a file name in angled brackets. For now, you simply need to know that this line tells the compiler to include other files. The inclusion of these files will allow us to use the functions they define. In this case, the `stdio.h` file defines the `printf` function used in Line 4. We will look at the preprocessor directives in further detail in Chapter 6.

Following this preprocessor directive, we come to the beginning our `main` function on Line 3. For the purposes of this course, all our programs will have a `main` function that looks exactly like this. This function is where the program begins its execution. You may notice the `argc` and `argv` variables in parenthesis. This indicates that the `main` functions expects two variables, `argc` and `argv`. The first (`argc`) should be of type `int` and the second (`argv`) should be of type `char**`. These variables are set by the system and we will ignore them for now.

You will also notice that Line 3 ends with an opening curly bracket ('{'). Further, you may notice that there is a closing curly bracket ('}') on Line 7. These brackets establish a code block and limits the scope of any variables used inside of them. Simply put, anything within these curly brackets are part of the function whose declaration they immediately follow (in this case, `main`). We will take a closer look at the `main` function and its structure in Chapter 3.

Within this main function, the first function to be called (and the first thing the program will do) is the `printf` function on Line 4. This function tells the system to print "Hello World!" to the output console. The "$\backslash n$" following the string is actually a representation of the newline character. That is, "$\backslash n$" tells the system to move to the next line such that any subsequent output will be printed on the new line. We will consider input and output further in Chapter 2.

On Line 6 we see a `return` statement. `return` statements tell the program to return from the function we are currently in. In this case, we are in the `main` function, thus a return will exit the program. However, in a more general case, a `return` statement will end execution of the current function and continue execution directly after the point the current function was called.

In our example, the `return` is followed by a 0. This indicates that the `return` statement should return the value '0' to the calling function. We know that this function must return an integer because the declaration of our `main` function on Line 3 begins with the `int` keyword.

Finally, you may notice that lines 4 and 6 end in a semicolon (';'). In C, all declarations and statements must end with a semicolon. This will quickly become second nature, but is also a common mistake for new programmers. If in doubt about an error, check that all your declarations and statements end with a semicolon.

This first section was meant to give you a very general overview into the C programming language. From here we will dive in a bit deeper and consider the individual constructs a bit more closely.

1.2. Comments

The first, very simple, construct I will describe in detail are comments. Comments are completely disregarded by the compiler, but are helpful for human understanding of code. Sometimes it is not immediately clear what a specific code section does. A simple comment might save someone reading your code valuable time. Comments are depicted in the following code listing.

```
 1 #include <stdio.h>
 2
 3 /*
 4 this is the main function
 5 program execution begins here
 6 */
 7 int main(int argc, char **argv){
 8   //the following line calls printf
 9   printf(”Hello World!\n”);
10
11   return 0; //this will return from the main function
12 }
```

Listing 1.3: A basic C program with comments

As can be seen in Listing 1.3, there are two types of comments. The first are **multi-line comments** as depicted in Lines 3 to 6. Such comments can span multiple lines and are enclosed in `/* ... */`.

In addition to multi-line comments, C support **single-line comments**. Such comments are preceded by `//`. That is anything on a line after `//` will be disregarded. We see this construct being used in two slightly different variations in our example. In Line 8, everything on the line is a comment and disregarded by the compiler. However, in Line 11 everything up to `//` is processed by the compiler and only everything after is disregarded.

1.3. Variables

You are hopefully familiar with the concept of variables from your algebra courses. In programming, the concept of a variable is very similar. We use variables as placeholders for information that will not be constant from run to run. However before we use a variable they must be declared so that the compiler is aware of them.

1.3.1. Declaration

For the purposes of this course a function declaration takes the form *type name;* where *type* refers to the variable type being declared and *name* refers to the name given to the variable. For example, the `int` type is likely the most common type you will come across in this course and it specifies that the declared variable is an integer. Here an example:

```
...
  int x;
  int y, z;
...
```

Listing 1.4: Integer declaration

Please be aware that the ellipses (...) do ***not*** *reflect a construct in the C programming language but are rather there to indicate this is just a snippet of code and there is code missing before and after the snippet.*

In Listing 1.4, we see three variables being declared as integers, `x`, `y`, and `z`. You may also notice that multiple variables of the same type can be declared on one line with one type specifier as is shown in Line 4.

1.3.2. Data Types

Of course, there are many types that can be specified in the C language. Below, I list the types that will be relevant for this course:

void This is special type specifier that indicates the type of this variable is unknown or bound to change. We will talk about the void type when we begin discussing pointers in Chapter 4.

char This type specifier indicates the variable is a single character (**not** a string).

int This type specifier indicates the variable is an integer. That is, a positive or negative whole number.

float This type specifier indicates the variable is a floating point decimal number. Essentially this type is a very precise approximation of real numbers.

In addition to these type specifiers, `int` and `char` types can be further specified with the `signed` and `unsigned` type specifiers beforehand. These allow the programmer to explicitly specify whether a variable can represent positive and negative numbers or simply positive numbers. For example, the following specifies `x` will hold only positive integers while `y` and `z` can be used to store positive or negative integers. Finally, `a` may only store positive characters.

```
...
  unsigned int x;
  signed int y;
  int z;
  unsigned char a;
...
```

Character Representation

The concept of a positive or negative character may be foreign to you, but if consider that the computer simply sees a number that represents a character, it may be easier to understand.

Internally, the computer stores all data as numbers. We simply have a standard for representing characters as numbers called the American Standard Code for Information Interchange (ASCII). This allows us to map numbers to characters in a standard way. For example, what we know to be the letter 'a' is stored by the computer as the number 97 (for the full ASCII table, see Appendix A). So really, the `char` type is an integer that is represented as characters when we input or output them. Hopefully, it is now clearer why we can distinguish between negative characters. Effectively, you can ignore the concept of unsigned characters for the purpose of this course, **however**, it is important that you understand that characters are internally represented as integers. For a took at the ASCII table, please see Appendix A.

We can even tell the computer to print an `int` as a `char` and if it is a valid character, the character will be printed, but more on that later...

Boolean Type

You may be wondering why I did not mention a boolean type. The answer is because there is no explicit boolean type in the C programming language. Instead we use integers when evaluating equality, for example. That is, an integer value of 0 is said to be false and all other integer values are said to be true. However, we generally stick with the values 0 and 1, specifically we use 0 to represent false and 1 to represent true. Additionally, if an expression evaluates to true, this generally means it evaluates to the integer value 1.

Note: You may see `true` and `false` used in a C program. These are however simply placeholders for 1 and 0 respectively to help readability.

1.4. Operators

This section will cover the basic operators you need to get a foothold in C. This will not be a complete list, but will give you a good start. For example, I will not cover bitwise operations and there are some pointer-related operators that we will cover later in Chapter 4.

1.4.1. Assignment

Likely, the single most used operator is the assignment operator. This operator is denoted with the equals sign ('=') and is used to assign a value to a variable.

```
1 ...
2   int x, y;
3   char z;
4 
5   x = 5;
6   y = x + 2;
7   y = y + 1;
8   z = 'q';
9 ...
```

Listing 1.5: Simple Assignments

In the example above, we see the integer variable `x` being assigned the value `5` in Line 5. In Line 6, we the integer variable `y` being assigned the value `x + 5` (in this case, `7`). In Line 7, we see an interesting though subtle peculiarity. It is important to note the difference between assignment and equality. In algebra, for example, the equals sign is used to denote equality, however in the C programming language it denotes assignment. Line 7 illustrates this point nicely. In an algebra sense, we would say `y` cannot equal `y+1`, therefore the statement is false. However, in C this is a statement assigning the value of `y+1` to the variable `y`. In essence, Line 7 increments `y` by 1.

Finally, in Line 7, we see the character variable `z` being assigned the value '`q`'. Notice that `q` is in single quotes. Single quotes denote a character and tell the compiler to treat it as such and not as a variable.

1.4.2. Arithmetic Operations

Arithmetic operators are equally as straightforward and are hopefully familiar from previous mathematics courses. These include addition ('+'), subtraction ('-'), multiplication ('*'), division ('/'), and modulo ('%').

```
1 ...
2   int x, y, z;
3
4   x = 5 + 2;
5   y = x + (2 / x);
6 ...
```

Listing 1.6: Arithmetic Operations

In the above example in Line 3, we see a simple addition taking place, which is then assigned to the variable `x`. In Line 5, we see that it is also possible to manipulate the order of operations with parenthesis. As in algebra, the operations within the parenthesis are evaluated first.

Modulo

I will assume that you are familiar with standard mathematical operations such as addition, subtraction, multiplication, and division. However, the modulo operation may be new to you. It is a very simple operation that denotes the remainder after the division of two numbers. That is, 5%2 = 1 because 5/2 is 2 with a remainder of 1.

Short-hand Assignment and Arithmetic

There are is some common short-hand syntax for combining assignment and arithmetic in C. While they are generally "short-hand" and are equivalent to their longer counterparts in their assignment behavior, they are so frequent that I will quickly list them here along with their "long-hand" equivalents.

`x++;` ≡ `x = x + 1;`

`x--;` ≡ `x = x - 1;`

`x+=y;` ≡ `x = x + y;`

`x-=y;` ≡ `x = x - y;`

`x*=y;` ≡ `x = x * y;`

`x/=y;` $\equiv$ `x = x / y;`

`x%=y;` $\equiv$ `x = x % y;`

1.4.3. Logical Operations

As mentioned, the C language does not have an explicit boolean type, however we can still perform logical operations on integers, keeping in mind that 0 is false and any non-zero integer is true. These operations include 'logical and' ('&&'), 'logical or' ('||'), and negation ('!').

```
1  ...
2    int x, y, z;
3
4    x = 1;
5    y = x || 0;
6    z = !y;
7  ...
```

Listing 1.7: Logical Operations

In the above example, we see on Line 4 that `x` is being assigned the value of 1 (i. e., true). On Line 5, `y` is assigned the value of `x` OR 0, which evaluates to 1 (i. e., true). Finally on Line 6, `y` is negated and the result is assigned to `z`.

1.4.4. Relational Operations

A relational operation is used to compare values and returns a boolean value. Since we do have boolean values at our disposal, in C these operations return either 0 (false) or 1 (true). The operators include 'equal to' ('`==`'), 'not equal to' ('`!=`'), 'greater than' ('>'), 'less than' ('<'), 'greater than or equal to' ('>='), and 'less than or equal to' ('<=').

```
1  ...
2    int x, y, z;
3
4    x = 1;
5    y = 5;
6    z = (x < 5);
7    z = (y != 5);
8  ...
```

Listing 1.8: Relational Operations

In this example, we see `x` and `y` being assigned integer values on Lines 4 and 5. We then see `z` being assigned the value of $(x < 5)$ on Line 6. If we assume that $x = 1$, we know that $(x < 5) =$ true, hence `z` in assigned the value 1 on Line 6. Further, `z` is assigned the value of $(y \neq 5)$ on Line 7. If we now assume that $y = 5$, we that $(y \neq 5) =$ false, hence `z` in assigned the value 0 on Line 7.

1.5. Control Structures

Control structures allow you to control the flow of the program based on some condition. These can generally be divided into two categories, branch structures and loops.

1.5.1. Branch Structures

A branch structure generally allows you to branch the control flow into two or more directions based on some condition.

`if-else` Statements

The most straightforward branch statement is the `if-else` control block and is best explained in a series of examples. First, the simplest case...

```
1  ...
2    int x;
3  ...
4    if(x < 10){
5      //do something
6      ...
7    }
8  ...
```

Listing 1.9: A simple `if` statement

In Listing 1.9, we see a very simple `if` statement. In this example, the code block from Lines 5 to 7 will be executed if the conditional statement on Line 4 (i. e., $x < 10$) is true. Notice the curly brackets that begin on Line 4 and end on Line 7. These enclose the code block that will be executed.

This example is extended below.

```
1  ...
2    int x;
3  ...
4    if(x < 10){
5      //do something
6      ...
7    }
8    else{
9      //do something else
10     ...
11   }
12 ...
```

Listing 1.10: A simple `if-else` statement

In the above example, we see the same `if` statement as in Listing 1.9, however it is followed by an `else` statement. The code block following this `else` statement (again, enclosed in curly brackets) is executed if the conditional statement on Line 4 is **not** true. That is, Lines 9 to 11 are executed if $\neg(x < 10)$. Finally, this example can be taken one step further.

```
...
  int x;
...
  if(x < 10){
    //code block A
    ...
  }
  else if((x >= 10) && (x < 15)){
    //code block B
    ...
  }
  else if((x >= 15) && (x < 25)){
    //code block C
    ...
  }
  else{
    //code block D
    ...
  }
...
```

Listing 1.11: A `if-else` statement

In this final example, we see a further construct, the `else if` statement. This construct allows us to perform multiple conditional tests with only one outcome. In Listing 1.11, for example, code block A is executed if $x < 10$, code block B is executed if $10 \leq x < 15$, code block C is executed if $15 \leq x < 25$, and code block D is executed otherwise (i. e., $x \geq 25$).

`switch` Statements

A very long `if-else` statement can become tedious or even inefficient in certain circumstances. For this reason, the C language also provides the `switch` construct for conditions that hinge on a single variable. For example, the following code is valid C code, but a bit tedious.

```
...
  int x;
...
  if(x == 1){
    ...
  }
  else if(x == 2){
    ...
  }
  else if(x == 3){
    ...
  }
  else if(x == 4){
    ...
  }
  else{
    ...
  }
...
```

Listing 1.12: Testing `x` with an `if-else` block

As mentioned the above example is completely valid, but there is a construct that may be better suited to such control needs. This construct is a `switch` statement. As usual, we will take a look at this in the context of an example. The following code snippet is semantically equivalent to Listing 1.12.

```
 1  ...
 2    int x;
 3  ...
 4    switch(x){
 5      case 1:
 6        ...
 7        break;
 8      case 2:
 9        ...
10        break;
11      case 3:
12        ...
13        break;
14      case 4:
15        ...
16        break;
17      default:
18        ...
19        break;
20    }
21  ...
```

Listing 1.13: Testing `x` with an `switch` statement

The syntax for a switch statement is a bit different and, as mentioned, it can only check the condition of a single variable or statement. On Line 4, we see the variable in question, in this case `x`, in parenthesis after the `switch` statement. This indicates that the variable `x` (this may also be a statement) will be used to test each condition. The `case` statements within the `switch` block are used to indicate the various possible values for `x`. That is, if $x = 1$, everything between Lines 5 and 7 will be executed, if $x = 2$, everything between Lines 8 and 10 will be executed, and so on. Finally if no conditions match, the `default` case is executed (everything between Lines 17 and 19).

One additional interesting point to remember is that each `case` must end with a `break` statement. This explicitly tells the program to continue execution after the switch statement. If the `break` statement is omitted, execution will continue **within** the switch block until a `break` is encountered or the block terminates. This allows us to create constructs such as the following:

```
 1 ...
 2   int x;
 3 ...
 4   switch(x){
 5     case 1:
 6     case 2:
 7     case 3:
 8       ...
 9       break;
10     case 4:
11       ...
12       break;
13     default:
14       ...
15       break;
16   }
17 ...
```

Listing 1.14: Testing `x` with an advanced `switch` statement

The above example still tests the value of `x`, but will now execute the code between Lines 7 and 9 if $(x = 1 \lor x = 2 \lor x = 3)$.

1.5.2. Loops

Loops are a control structure designed to repeat a set of statements repeatedly while some condition holds. We will consider two forms of loops, namely `for` loops and `while` loops.

`while` Loops

`while` loops are a very basic form of loops that simply repeat a code block while some condition holds and take the following form:

```
1 ...
2   int x;
3 ...
4   x = 1;
5   while(x){
6      ...
7   }
8 ...
```

Listing 1.15: A simple `while` loop

In the above example, the code block from Line 6 to 7 is repeated as long as the condition after the `while` keyword on Line 5 holds. Specifically, this loop will be repeated as long as $x = \text{true}$. Remember that boolean values are simply integers in C. So using `x` alone as a conditional is completely valid.

`for` Loops

`for` loops are a bit different in their syntax and are generally suited to facilitate iteration over a variable. Their general form is:
`for(`[initialization]`;`[loop condition]`;`[update statement]`)`
Again, we will look at an example.

```
...
  int x;
...
  for(x=0;x<20;x++){
     ...
  }
...
```

Listing 1.16: A simple `for` loop

The `for` loop in this example initializes `x` to 0 ([initialization]), loops as long as $x < 20$ ([loop condition]), and increments `x` by 1 at the end of each iteration ([update statement]). Simply put, this loop iterates 20 times and increases the value of `x` by 1 for each iteration.

`break` and `continue`

There are two more interesting keywords in the C language when speaking about loops. These are `break` and `continue`. Both of these keywords will interrupt the current loop iteration immediately. The difference between the two lies in where execution continues after the statement.

The `break` keyword will immediately break out of the current loop and continue execution after the current most nested loop. Below is a simple example.

```
 1 ...
 2   int x;
 3   int y;
 4 ...
 5   for(x=0;x<20;x++){
 6     ...
 7     if(y){
 8       break;
 9     }
10   }
11 ...
```

Listing 1.17: A simple `for` loop with a `break`

The above example makes use of the same `for` loop as in Listing 1.16 with the exception of the `break` keyword. The loop in this example will run in the same manner as in Listing 1.16 however, if at any point the execution hits Line 7 and y = true, the execution will break out of the `for` loop and continue on Line 11 regardless of the current state of `x`.

Similar to the `break` keyword, the `continue` keyword will stop the current loop iteration. However, instead of continuing execution after the loop, execution will continue with the next iteration of the loop regardless of whether there is still code to be executed in the loop block. We will take a look at another example.

```
 1 ...
 2   int x;
 3   int y;
 4 ...
 5   for(x=0;x<20;x++){
 6     ...
 7     if(y){
 8       continue;
 9     }
10     ...
11   }
12 ...
```

Listing 1.18: A simple `for` loop with a `continue`

In the example above, the loop will execute as normal however, if at any point the execution hits Line 7 and y = true, the execution will continue with the next iteration of the loop (i. e., `x` will be incremented) regardless of the fact that there is still code in the loop block between Lines 9 and 11. This code will not be executed for this iteration.

2. Organizing and Outputting Data

This chapter will cover structures, arrays, strings, and the most common user input and output functions.

2.1. Structures

Structures (sometimes simply referred to as "structs") are a simple way of organizing data into groups. We already spoke about variables and variable types, however sometimes we may have the need to represent data that requires multiple variables. For example, a three-dimensional point in a euclidean space is generally represented by three coordinates (`x`, `y`, and `z`). We could of course represent two such points with the following variables:

```
...
  float point_a_x;
  float point_a_y;
  float point_a_z;
  float point_b_x;
  float point_b_y;
  float point_b_z;

  point_a_x = 0.5;
  point_a_y = 1;
  point_a_z = 2;
  point_b_x = 13.22;
  point_b_y = 17.2;
  point_b_z = -12;
...
```

Listing 2.1: Declaring and assigning point variables

However, you may notice that each such point that we want to represent will have the same three variables (`x`, `y`, and `z`). A struct will allow you to organize this data. To define a struct, we use the following syntax:

```
...
struct point {
  float x;
  float y;
  float z;
};
...
```

This will define a group of variables that belong together. Once a struct has been defined, a struct variable can be declared with the following syntax:

`struct point a;`
Once this variable has been declared, `a` is a variable that effectively contains three float variables. These can be individually addressed with a dot ('.') as follows:
```
a.x = 0.5;
a.y = 1;
a.z = 2;
```

If we now recreate what we did in Listing 2.2 using structs, it would look like this:

```
...
struct point {
  float x;
  float y;
  float z;
};
...
  struct point a;
  struct point b;

  a.x = 0.5;
  a.y = 1;
  a.z = 2;
  b.x = 13.22;
  b.y = 17.2;
  b.z = -12;
...
```
Listing 2.2: Declaring and assigning point variables using a struct

You can see that this will save you a bit of typing and sanity if the program requires you work with multiple points.

2.2. Arrays

An array is a fairly simple concept. Imagine you are given the task of storing all the test scores of the students in this lecture for some task. Assuming there are 150 students and each is assigned an identifier from 0 to 149, the following would be a correct way of accomplishing this task:

```
...
  int student000;
  int student001;
  int student002;
  int student003;
...
  int student147;
  int student148;
  int student149;
...
```

While correct, you may agree that the above method is rather tedious. Declaring 150 variables like this seems silly. Luckily the C language offers a solution, namely arrays. Arrays are simply arrangements of some size of the same variable type. The syntax makes use of square brackets to define the size of the array. The following example declares the same 150 integer variables as in the above example:

```
...
  int student[150];
...
```

Clearly this seems much easier. The question remains as to how one addresses the individual integers. This is just as simple and uses the same square brackets. Lets look at the following example.

```
...
  int i;
  int student[150];
...
  for (i=0;i<150;i++){
    //get grade for current student and set the variable cur_grade
    ...
    student[i] = cur_grade;
  }
...
```

Listing 2.3: Assigning values to an array of integers

We know from our experience with `for` loops in Section 1.5.2, that `i` has an initial value of 0 and is incremented by one in each iteration. This means that in the first iteration of the loop, we set the value of `student[0]`, in the second iteration, we set the value of `student[1]`, and so on. You will also notice that the index of the array (the number inside of the square brackets) starts with 0 and **not** with 1. This is a very important point that is often overlooked by new programmers. This also means that the 150th element has an index of 149.

Important: It is up to you, the programmer, to know how large your arrays are. If you declare an array as such,
`int a[10];`
it is up to you to know that this array holds ten elements (index 0 through 9), if you address the 15th element with,
`a[14] = 100;`
the compiler will **not** stop you. It will assume you are a competent and smart programmer who knows what he/she is doing. However, this is will likely lead to very bad things, so be careful!

In addition to addressing individual elements of the array with square brackets, you may refer to the entire array by leaving the brackets away completely. This will result in the array variable type “decaying” to become a pointer. This may be useful when passing arrays to functions, for example. However, we will consider this more closely in the next chapters.

Finally, the following may seem like correct C code:

```
 1  ...
 2    int a[10];
 3    int b[10];
 4    int i;
 5
 6    for(i=0;i<10;i++){
 7      a[i] = i;
 8    }
 9
10    b=a;
11  ...
```

Listing 2.4: An incorrect array copy

However, the compiler will complain with an error on Line 10. This is **not** the proper way to copy arrays. Among other things the compiler doesn't know how long each of these arrays is and therefore cannot copy them. The proper way to copy array `a` to array `b` is as follows:

```
...
  for(i=0;i<10;i++){
    b[i] = a[i];
  }
...
```

Listing 2.5: A correct array copy

This explicitly copies a number of elements from one array to another. Thus, it is once again up to the programmer to know how big the arrays are.

2.3. Strings

Strings are nothing more than a special form of arrays. I already introduced the character data type in Section 1.3.2. Strings are nothing more than arrays of characters with one exception. They are specially terminated. As I mentioned in the previous section, the compiler has no idea how long an array is, let alone how long a string inside of an array is. Think of a character array as a "container" for a string. A character array of size 32 can contain a string of size 31 (remember, there must be room for a special terminating character), but it can also contain any string with fewer than 31 characters. So the compiler must know where the current string stops when printing it, for example.

Let us consider the following example:

```
...
  char s[12] = "long string";

  strncpy(s, "short", 5);
...
```

Listing 2.6: A string example

This example assigns a string variable `s` the value `''long string''`, then immediately reassigns this variable with the value `''short''` in Line 4 (we will get to the `strncpy` function below, for now know it copies "short" into `s`). Let us assume there are no terminating characters and keep in mind that the compiler doesn't know how long the string is. If we look into the array, as the computer sees it it would look similar to Figure 2.1.

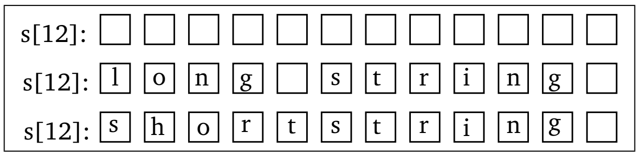


Figure 2.1.: String assignment without a terminating character.

Considering, Figure 2.1, if we ask the program to output `s`, how should it know what to output? Without any terminating characters, the program may end up up outputting `''shortstring''` after the reassignment on Line 6. However, what we want is that the program simply output `''short''`. This is where the terminating character comes in. This terminating character is referred to as `NULL` and when represented as a character is represented as '\0'. The compiler (or in some cases, a function) will automatically append this `NULL` byte to the end of a string. This is why we must also always make sure our array is at least one character larger than the maximum size string the array is to hold.

If we consider a proper copy, this time with a terminating character, it would look as it does in Figure 2.2.

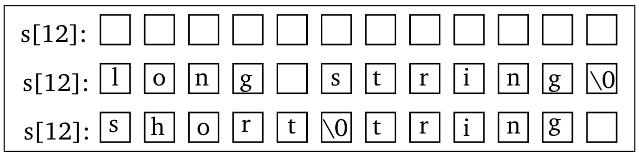


Figure 2.2.: String assignment with a terminating character ('\0').

Now, it is clear to the program what to output when asked. It simply prints characters until it comes across the `NULL` byte.

Important: Keep in mind that after your string there are likely other variables or data stored in memory. Additionally, neither the compiler nor the underling hardware know how long your array or string is. So if you ask your program to print a string without a terminating character, your program will output **all data in memory** until it hits a `NULL` byte by chance. This is not good!
However, it is generally the case that you will not have to worry about this as most string operations and functions will append the `NULL` byte automatically.

Important: I will reiterate how important it is to always leave one character at the end of your array free for the `NULL` byte. If you do not do this, the compiler or function may automatically write a `NULL` byte at the end of your string. However, if this no longer is within the bounds of your array bad things will happen!

2.3.1. `strlen` Function

The next three subsections will cover common string functions that you might find useful. The first of these is the `strlen` function. You may be able to guess from the name that this function returns the size of a string as an unsigned integer. Specifically, this function takes a single argument that is the string for which you wish to know the length and returns an unsigned integer that represents the size of that string. It is very important to note that this function returns the size of the `string` and not the size of the `array`. That is, this function simply iterates over the string until it finds a `NULL` byte and outputs the number of characters before it. As always, the compiler and machine do not know the size of your array, it can only tell you the size of the string within the array. Below is an example:

```c
1 #include <string.h>
2 ...
3   char s[100] = "this is a string of length 29";
4   unsigned int size;
5 
6   size = strlen(s);
7 ...
```

Listing 2.7: `strlen` in action

You will notice on Line 1 that the `string.h` header must be included in order to use this function. On Line 6, `strlen` is used to assign the size of the string `s` to the unsigned integer `size`. After Line 6 is executed, `size` will contain the value 29. Notice that it contains 29 and **not** 100 as the size of the string is 29 even though the size of the array is 100 as declared on Line 3.

2.3.2. `strncpy` Function

One common mistake that new programmers make when using C is they want to copy strings with the following:

```
...
  char s[20];
  char d[20];

  s = "string to be copied";

  d = s;
...
```

Listing 2.8: The incorrect way to copy a string

This will result in an error. I will hold off on explaining why this causes an error until we have discussed pointers. For now you simply need to know that this is the wrong way to copy strings. The correct way to copy strings is either to assign it directly when declaring the string or to use a function called `strncpy`. This function takes three arguments. The first is the destination string (the string to which we want to copy), the second argument is the source string (the string from which we want to copy), and the third argument is the size of the destination array. It is important that the size argument is correct as it may lead to an overflow if the destination is actually shorter than the size you give the function. Also, beware if the destination string is smaller than the source string. If this is the case, the function will copy as much of the string as possible **without** copying a `NULL` byte. So if this is your intention, you need to manually add the `NULL` byte to the end of the string yourself. Below is an example of how to properly copy a string:

```
1 #include <string.h>
2 ...
3   char s[20] = "string to be copied";
4   char d[20];
5
6   strncpy(d,s,20);
7 ...
```

Listing 2.9: The correct way to copy a string

The first thing you may notice in the above example is that `string.h` is included on Line 1. As with the `strlen` function this header must be included in order to use this function. Additionally, you will notice that I am assigning the source string directly on Line 3 as part of the declaration. The actual copying is quite straightforward. On Line 6 you see the proper way to copy a string using the `strncpy` function as described above.

Important: There is a function similar to `strncpy` called `strcpy` (notice the lack of the 'n'). This function works without the size argument and simply copies one string into another until it reaches a `NULL` byte in the source string. This function is considered unsafe as it may result in an overflow of the destination. Remember what I told you regarding strings and arrays! Neither the compiler nor the underling hardware know how long your array or string is. It is up to you to write your code such that it does not overflow!

2.3.3. `strcmp` Function

As with copying strings, new C programmers often try to compare strings with the following:

```c
1 ...
2   char s1[20] = "string1";
3   char s2[20] = "string2";
4
5   if(s1 == s2){
6 ...
```

Listing 2.10: The incorrect way to compare two strings

What makes this even more confusing than the example I used with copying strings is that this code will compile **without any errors**! However, Line 8 in the above example is **not** comparing the strings. Again, I will forgo an explanation until we have discussed pointers, but know that this is incorrect if you want to compare the two strings. The proper way to compare strings is to use the `strcmp` function. This function simply takes two strings as arguments, compares them, and returns an integer result. Specifically, it returns 0 if the two strings are the same and it returns a non-0 value if they are not. This may seem counterintuitive since we learned that non-0 integer values equate to `true` and 0 equates to `false` in a boolean sense. However, the non-0 value that is returned allows you to infer something about how the strings are different that are irrelevant for this course. Below an example:

```c
1 #include <string.h>
2 ...
3   char s1[20] = "string1";
4   char s2[20] = "string2";
5
6   if(strcmp(s1,s2) == 0){
7 ...
```

Listing 2.11: The correct way to compare two strings

As with all the string functions, this function requires `string.h` to be included as shown on Line 1. The `strcmp` function is used on Line 6 and its return value is compared to 0 in order to determine if `s1` and `s2` are equal.

2.4. Console Output

We will use one primary function for outputting information to the screen in our programs. This is the `printf` function. As breifly mentioned in Section 1.1, this function requires that `stdio.h` is included. This is done at the beginning of the program with the following syntax:
`#include<stdio.h>`
Such "include statements" can generally be the first things in your source file.

Printing a constant string is very straightforward with the `printf` function, as can be seen in the following:

```
#include<stdio.h>
...
  printf("Hello World!\n");
...
```

Listing 2.12: Printing a constant string.

The above example will print the string "Hello World!" to the console. Remember, that "\n" is a representation of the newline character as introduced in Section 1.1. That is, this sequence will simply move the cursor to the next line.

Printing variables is a bit more complex, but still rather straightforward. The `printf` function uses a concept called **format strings**. The idea is fairly simple. For every variable you want to embed in the output string, put a placeholder in the string. Imagine we use '%' as our placeholder and we want to embed a single variable in our output. The string may look something like this: `''Hello, I am % years old.\n''`. The compiler will then know that it is to print the string, but replace all placeholders with the respective variable(s). You may ask, how does the compiler know what variable to replace the placeholder with? Well you simply add the variable as an additional variable as such:
`printf(''Hello, I am % years old.\n'',age);`
where `age` is a variable that stores the age to be output. If I want to output additional variables, I simply add those variables as arguments in the order I want the replacement to occur, as such:
`printf(''Hello, I am % years old and my name is%.\n'',age,name);`
In this case, `age` is a variable that stores my age and `name` is a variable that stores my name. Notice that the order of the variables reflects which variable replaces which placeholder. That is, the first variable will replace the first placeholder, the second variable will replace the second placeholder, and so on.

There is one final step that is still missing. As you may have noticed by now, the compiler and the machine itself are very primitive and need to be told everything. This means you need to explicitly state what variable **type** will be replacing the placeholder. This is done with a letter after the '%'. For example, if the placeholder is to be replaced by an signed integer, the placeholder is "%d" and if the placeholder is to be replaced by a string, the placeholder "%s" is used. So our call to `printf` would look as follows:
`printf(''Hello, I am %d years old and my name is%s.\n'',age,name);`
This is due to the fact that `age` is an integer variable and `name` is a string variable.

A table with placeholders relevant for this course follows:

Placeholder	Type
%d	signed integer
%u	unsigned integer
%f	floating point
%c	single character
%s	null terminated string

Important: Format strings can be dangerous. As shown, the first argument must always be a string to be output. This means that the following will compile and run with seemingly no problem:
`printf(name);`
where `name` is a variable containing a string. This can be very dangerous and lead to a vulnerability if the user can influence the `name` variable in any way (which is generally the case). The class of exploit that abuses this vulnerability are called **format string exploits**. I will not go into details, but know they exist. The proper way to output a single string is as follows:
`printf(‘‘%s’’,name);`
This may seem redundant, but is very important!

2.5. Console Input

Accepting input from the user is very common place in most programs. One of the most basic methods for accepting input is directly from the console. For now, we will learn to accept all input as a string first. This is very straightforward with the `fgets` function. As always, I will explain with an example.

```
1 #include<stdio.h>
2 ...
3   char input_string[128];
4   fgets(input_string, 128, stdin);
5 ...
```

Listing 2.13: Accepting input

The first thing you may notice is that the same header that was included above for `printf` is included here on Line 1. Also, we must first create the string that is to contain the input the user types. On Line 3, you see we declare a string of size 128 to hold the input. Finally, on Line 4 the input is actually collected with the `fgets` function and stored in the `input_string` variable. Let us take a closer look at `fgets`.

`fgets` accepts three arguments. The first is the string variable that is to hold the input (`input_string` in the example). The second is the size of the string passed as the first argument (128 in the example). Finally, the third argument is the file descriptor of the input stream (`stdin` in the example). For our purposes, the first two are of most interest. It is very important that the second argument correctly reflects the size of the first argument. As I have reiterated several times, neither the compiler nor the machine know (or care) how large your arrays are. Therefore, you must explicitly tell the function how large the array is. If you tell `fgets` that the input string is larger than it actually is, it will happily write past the string buffer in memory. If this occurs, bad things will happen.

The final argument will always be the same for this purpose. `stdin` is the file descriptor for keyboard input and `fgets` is a generic function that can get input from any open file. In Linux, everything is handled as a file and reading and writing from/to the console is the same (for the programmer/user) as reading and writing from/to a file. In this case, `stdin` simply tells the function to read from the keyboard (and not from a “actual” file).

Finally, it is important to understand what `fgets` actually does. There are some questions you might be asking yourself: Does the user have to enter exactly 128 characters? If not, how does the function know to stop getting characters early? Does the `fgets` function automatically add the `NULL` terminator? Well, first of all, the user does **not** have to enter exactly 128 characters in the above example. **fgets** will accept input until the the buffer is full **or** a newline (*enter* key) is received from the user. So, if a user enters 5 characters, then presses *enter*, the input string will contain the 5 characters entered followed by a `NULL` terminator. This answers the third question as well, the `fgets` will automatically `NULL` terminate the string. So, this actually means that if the second argument is 128 as in our example, the function will only receive a maximum of 127 characters as it saves space for the `NULL` terminator.

Important: There are many ways to accept input into a string using C. However, it is important that you the programmer **ALWAYS** make sure that you are able to control the amount of input received based on the size of your buffer. This is such a critical point as it results in a most basic vulnerability. You should never allow a user to enter input past the bounds of an array. This will almost always result in a vulnerability that can be exploited by a user.
There exist functions that do not allow you as the programmer to pass the size of the buffer and simply stop collecting input once the user presses *return*. These functions are **extremely dangerous** and should **never be used**. These functions are deprecated, but you must still be careful. If you ever think there might be a good reason to use such a function in C, you are wrong! Never use them.
Never.

Important: Really, never.

Converting Strings to Numbers

One point that remains is that with `fgets` we only can accept strings. That is, a user can of course enter “74863”, however the machine stores this as a string and not as an integer, which is what you may want. This means, we must take one additional step to convert the string “74863” to the integer 74863. Luckily, we have a function at our disposal to do this for us, namely the `atoi` (**a**scii **to** **i**nteger) function. The `atoi` function accepts a single string variable and returns an integer. See the example below.

```
 1 #include<stdlib.h>
 2 #include<stdio.h>
 3 ...
 4   char istr[128];
 5   int iint;
 6 
 7   fgets(istr,128,stdin);
 8 
 9   iint = atoi(istr);
10 ...
```

Listing 2.14: `atoi` in action

I will begin by pointing out that this function requires you include `stdlib.h` as can be seen on Line 1.

In the example above, you see on Line 9, the string `istr` being converted to an integer and stored in `iint`. You will need to exercise caution as this function will not validate the input. That is, if the input string does not represent a decimal number, the function will generally return 0. Specifically the function will ignore all leading whitespace (e. g., spaces and tabs), then start processing decimal digits character by character. If the first character is not a decimal digit, the function simply returns 0. If the first character(s) is/are decimal digits, it will process them until there are no more. That is, if the user enters “123abc567”, the function will return 123.

Additionally, there exists a similar function for floating point numbers, namely `atof` (**a**scii **to** **f**loat). This function works in exactly the same manner except that it returns a floating point rather than an integer.

3. Functions

Until now, we have been organizing our code into a single `main` function that is called automatically and using functions that have been predefined and implemented for us. In this section we will discuss declaring, implementing, and using our own functions. First let us talk about why functions are helpful and useful. Most programs **could** be written in a single function, however it would lead to code that is difficult to read, difficult to understand, and unnecessarily long. Let us assume there was a task that you found yourself needing several times in a single program. As a simple example, let us assume this task was finding the mean of three floats. This is simply calculated with the following C code:

```c
...
  float a;
  float b;
  float c;
  float result;
  ...
  a = 1.2;
  b = 34.2;
  c = 12.444;
  result = a + b + c;
  result = result / 3;
...
```

Listing 3.1: Calculating the mean of three floats

While this is a very simple example, I hope you can see that if you had to perform this task several times, it may make sense to write the code once and reuse it each time you needed it, rather than copy and pasting this code segment everywhere you needed it. This has the added benefit that if you realize halfway through your program that you actually need the median of the three numbers and not the mean, you simply need to change the code once rather than finding all the points in your code where you calculate the mean and change it.

3.1. Declaring Functions

Function declarations generally have three main parts, first, the **return value**. This declares the type of the data that will be returned by the function. Next is the **function name** itself. Similar to a variable name, you can arbitrarily name your function when declaring it. Finally, you must specify the **function arguments**. This is a list of variables that will serve as the input to the function.

After the declaration itself, you can immediately implement the function within curly brackets. Let us look at an example:

```
 1 ...
 2 float mean(float a, float b, float c){
 3   float result;
 4
 5   result = a + b + c;
 6   result = result / 3;
 7
 8   return result;
 9 }
10 ...
```

Listing 3.2: A function declaration

In the above example on Line 2, you see the declaration itself. The first keyword, `float` identifies the return value of the function. In this case the result (i. e., the mean of the three numbers) will be a float. This value is returned within the function with the `return` keyword. You can see this on Line 8. After the `float` keyword on Line 2, you see the function name, `mean`. This simply defines the name of the function such that it can be called and used later. Following this in parenthesis, you see a list of variable declarations separated by commas. This defines the input to the function and the names of those variables within the function. You will notice, on Line 5, the variables `a`, `b`, and `c` being used seemingly without declaration. However, the variable declaration within the function arguments is enough in this case. Additionally, the parenthesis may remain empty if there is no variables to be passed to the function.

After the declaration, you see the implementation of the function within curly braces. This should look very similar to the implementation used in Listing 3.1, except that it is encapsulated in a function.

As a final note, you may wish to declare a function that does not return any value. If this is the case, the return type should be `void`. For example, the following function declaration identifies a function that does not return any value:
`void function_name()`
In this case, the `return` statement may be used without a value following it to immediately return from the function.

3.2. Using Functions

Using functions is very straightforward and is something you have already done with functions that are implemented for you (e. g., `printf`).

Assuming that the `mean` function is declared and implemented (as in Listing 3.2), the code in Listing 3.1 could be rewritten as:

```
 1 ...
 2   float a;
 3   float b;
 4   float c;
 5   float result;
 6   ...
 7   a = 1.2;
 8   b = 34.2;
 9   c = 12.444;
10   result = mean(a, b, c);
11 ...
```

Listing 3.3: Using the `mean` function

In the above example you see the `mean` function we declared in Listing 3.1 be put to use on Line 10. Keep in mind that the function you want to use should be declared **before** you use it.

3.3. The `main` Function

Now that we know what a function declaration looks like, we will take a closer look at the `main` function and its declaration. For reference, the declaration looks as follows:
`int main(int argc, char **argv)`
You will notice immediately that the `main` function returns an integer. It is general convention that a successfully executed program will return 0, while an unsuccessful execution will return a non-0 value. How "successful" and "unsuccessful" are defined and what the non-0 values represent are left a bit to the programmer and should be documented.

You will further notice that the `main` function takes two arguments, `argc` and `argv`. Up to now, we have not touched these, but we will take a closer look now. `argc`'s type is straightforward, this argument is an integer. `argv`'s type declaration may look a bit scarier. This is actually a double pointer to a character, but for now you can simply consider `argv` to be an array of arrays. Specifically, `argv` is an array of strings, hence the `char` type. We will consider pointers in more detail in the next section.

You may be asking, "What do these variables actually contain if I never call main?". In fact, `main` is called by the underling system and will populate these variables with the **command line arguments**. You may have noticed that command line programs can be called with additional information on the command line after the name of the executable. For example, `ls -la /etc/` calls the program `ls` with additional information. You as the programmer must be able to retrieve this information within the program.

It is this information that is present in `argc` and `argv`. Specifically, `argc` is an integer that contains the number of arguments passed. This includes the executable itself. `argv` is an array of strings containing the arguments themselves. That is, the string that is typed into the command line is divided by spaces and each word or string is separated into an individual string.

Let us consider an example. Assume that your program, `cl` is called as follows:

```
$ cl arg1 arg2
```

In this case, `argc` contains the value 3 as there are three separate arguments on the command line (including `cl`). Additionally, `argv` is an array of length 3 (`argc`) of strings that contain the arguments. Specifically, our example array would look as follows:

```
argv[0] = ‘‘cl’’
argv[1] = ‘‘arg1’’
argv[2] = ‘‘arg2’’
```

Now with this information you can handle the command line arguments within your program.

4. Pointers & Type Casting

This chapter introduces the concept of pointers and type casting which is where C can get a bit confusing to the newcomer. Before we get into the details, I will cover a bit of background.

4.1. Memory Basics

The memory in which your program resides is simply an array of 8-bit bytes. Further, through the magic of modern operating systems (OSs), you can assume that you have the maximum amount memory possible (despite the amount of physical memory in your PC) and you don't need to worry about sharing it with other programs (assume you are the only program running in memory). As with an array, each byte of memory has a numeric index or **address** starting at 0. In a 32-bit machine this address is 32 bits long, while in a 64-bit machine this address is 64 bits long.

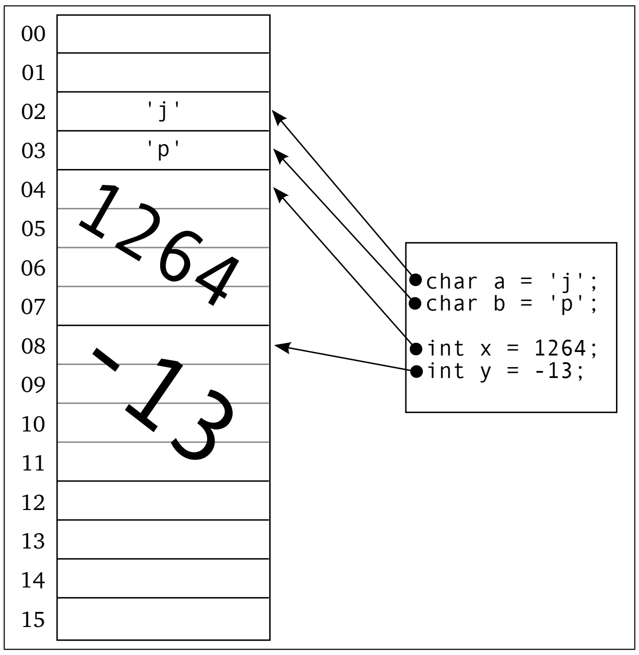


Figure 4.1.: Memory layout of variables

So each variable that is stored in memory has a value, but also has an address. Consider Figure 4.1. This figure assumes that characters are 8 bits in size and integers are 32 bits in size. Here you see four variables, two characters (`a` and `b`) and two integers (`x` and `y`). The value of `a`, for example, is `'j'` and the address of `a` is 02. Analogous, the value of `x` is 1264 and the address of `x` is 04. You will notice that since integers have a size of 32 bits, each integer takes up 4 entries in the array.

So, while it is possible to pass around the value of a variable, it is equally possible to pass around the address of a variable. A variable that contains the address of another variable is called a **pointer**. It is a variable that "points" to another variable, hence the name pointer.

4.2. Pointer Basics

Now that we know that a pointer is simply a variable that contains the address of another variable, hopefully some of the anxiety has died down. A pointer variable is identified at the time of declaration with an asterisk ('*') after the type. We still need to declare the type, so that the compiler and system know what variable type the pointer is pointing to. Consider the following declaration:

```
int *int_pointer;
```

The above declaration declares a pointer to an integer and has the name `int_pointer`. A pointer to any other type (including structs) can be declared in the same manner.

4.3. Referencing Variables

Now that we have pointers, we also need a way to reveal the address of variables. Assume that you have a variable whose address you wish to assign to a pointer. In order to reveal the address of a variable, we use an ampersand ('&'). Consider the following example:

```
1 ...
2 int x;
3 int *x_pntr;
4
5 x = 5;
6
7 x_pntr = &x;
8 ...
```

Listing 4.1: Variable reference example

In this example, we see an integer declaration on Line 2 and an integer pointer declaration on Line 3. On Line 5, `x` is assigned the value of 5. Finally, on Line 7 the pointer variable `x_pntr` is assigned the address of the variable `x`. Just to drive this point home, let us look at one more figure that visualizes what memory looks like.

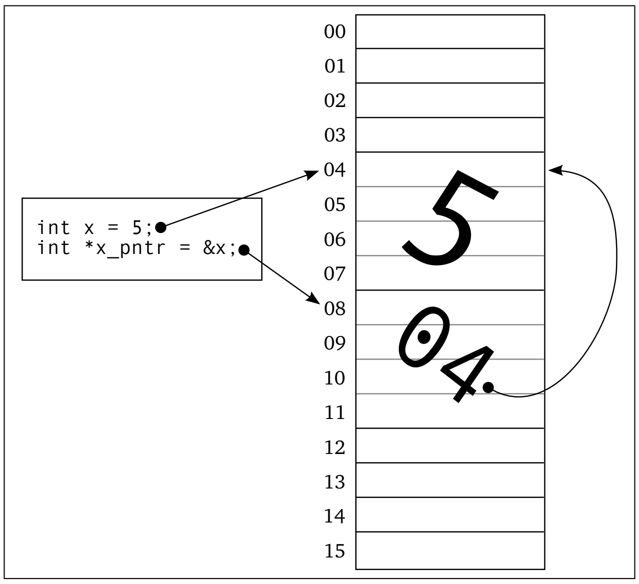


Figure 4.2.: Pointers in action

In the above figure, we again assume that integers are 32 bits in size. We also assume that pointers are 32 bits in size (32-bit architecture). As in Figure 4.1, we see the integer variable x assigned the value 5. Additionally, we see the integer pointer variable x_pntr being assigned the address of variable x. Since x is stored at address 04, this value is stored in the memory assigned to x_pntr.

4.4. Pointer Dereferences

In the same way that we may be interested in revealing the address of a variable, we may also be interested in revealing the value of a variable that a pointer points to. This is again done with the asterisk ('*'). The difference is that instead of using the asterisk during declaration, we prepend it to a variable name that has already been declared. Consider Figure 4.2 again and assume that we want to output the value of x through the pointer x_pntr. If we were simply to output the value of x_pntr, we would get 04 as this is what is stored in memory there (the address of x). If we want the value of what is stored at address 04, we use the asterisk in the form *x_pntr. This is illustrated in the following example:

```
...
int x;
int *x_pntr;

x = 5;
x_pntr = &x;
```

```
7
8 printf("x_pntr references the value %d\n",*x_pntr);
9 ...
```

Listing 4.2: Following a pointer reference

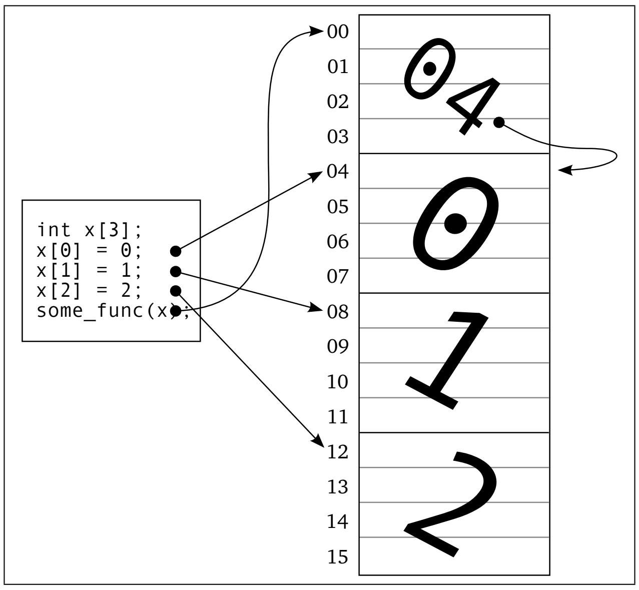
Listing 4.2 is identical to Listing 4.1 except for the call to `printf` on Line 8. This call will result in the string, "*x_pntr references the value 5*" being printed as `x_pntr` is prepended by an asterisk when passed to `printf`.

4.5. Arrays and Pointers

Up to this point, we have spoken about arrays and pointers separately. However, they are related such that it makes sense to discuss the interplay between them. In Chapter 2, I mentioned that a variable that is declared as an array "decays" into a pointer when you leave the brackets away. More precisely, this is true when the variable is used on the right-hand side of an expression or as input to a function. Additionally, any pointer can be used as an array by adding the bracket notation to it.

If you declare an array of integers as follows:
`int intarray[10];`
you can access the individual elements with brackets as discussed in Chapter 2 (e. g., `intarray[2]`). When we pass an array to a function (as we saw when discussing the string functions in Chapter 2), for example, we leave the brackets away completely (e. g., `somefunction(intarray);`). Effectively, what we are doing is passing a pointer to the array to the function. This pointer points to the first element in the array and since each subsequent element is stored sequentially in memory, they can be accessed within the function as well.

The reverse is true too. If we have a pointer, which is what we receive within a function, for example, we can access subsequent elements in memory simply by adding brackets. Consider the following diagram:



```
int x[3];
x[0] = 0;
x[1] = 1;
x[2] = 2;
some_func(x);
```

Figure 4.3.: Pointers and arrays

Here, we see that `x[3]` is declared as an array of size 3 and each element is assigned a value. You will also notice that the elements of `x` are stored sequentially in memory one after the other. When I pass `x` to the function `some_func`, I pass it without any brackets. This essentially passes a pointer to the first element in the array to the function. If I now (within `some_func`) begin using bracket notation again, the compiler will know to automatically dereference the pointer and access the proper index. Let us consider an example:

```
 1  ...
 2  void some_func(int *x){
 3    printf("%d\n",x[1]);
 4    return;
 5  }
 6
 7  int main(int argc, char **argv){
 8    int x[3];
 9
10    x[0] = 0;
11    x[1] = 1;
12    x[2] = 2;
13
14    some_func(x);
15  }
```

Listing 4.3: Pointers and arrays

You will notice on Line 2 that `some_func` is declared to accept a `int*` (i. e., a pointer to an integer), however on Line 3, we use the bracket notation to access the second element within the original array. That is, the `printf` function on Line 3 will output “1” when called from Line 14. The compiler automatically dereferences your pointer and jumps ahead one to access the 1 in memory.

In class exercise: Explain why string copies and compares need special functions as described in Chapter 2.

4.6. Type Casting

Type casting is a construct that allows you to access a variable as if it were of a different type. Some care must be taken when performing a type cast, but the principle is very simple. Let us assume that you have an integer variable `x`, however you have a function that requires you pass it a floating point decimal. This should be possible, an integer 15 can be expressed as a floating point 15.0 without issue, however we need to explicitly tell the compiler this is what we want to do. This is done by prepending the variable in use with the type to be cast to in parenthesis. Consider the following example:

```
 1  ...
 2  void print_float(float x){
 3    printf("%f\n",x);
 4    return;
 5  }
 6
 7  int main(int argc, char **argv){
 8    int x;
 9    float y;
10
11    x = 5;
12
13    y = (float) x;
14
15    some_func((float)x);
16    some_func(y);
17  }
```

Listing 4.4: Simple type casting

In the above example we see two examples of type casting, once on Line 13 and one on Line 15. On Line 13, we see the variable `x` being type cast to a float before its value is assigned to `y` and on Line 15, we see he variable `x` being type cast to a float before it is passed to a function. It is important to note that the type cast **does not** change the type of `x`, x remains an integer variable. Only its value is accessed as a floating point value on Lines 13 and 15.

It is also important to note that the type cast only affects the expression immediately following it. So `(float)x + y` is different from `(float)(x + y)` in that the former casts the value of `x` to a float, then adds that value to `y`, while the latter adds `x` and `y`, then casts the result of that addition to a floating point value.

Finally, it is very important to keep the size of variables in mind. Remember that I told you in Chapter 1 that characters are actually represented as numbers and integers can be printed as numbers. Well if you refer to the ASCII table in Appendix A, you will see that the letter 'a' is represented by the number 97. In fact you can print the letter a to the screen with the following:

```
...
  int x = 97;
  printf("%c",(char)x);
...
```

The above example will print the character 'a'. This is all well and good, however you must be careful because characters are stored with 8 bits, while integers are stored with 32 bits (this can vary depending on architecture, but let us assume 32 bits). This means that a character has a potential maximum value that is much smaller than that of an integer. Let us consider decimal numbers. If I tell you that you only have two digits available to represent a decimal number, you know that maximum possible value you can store is 99. On the other hand if I tell you that you have four digits available, you can store a maximum value of 9999, a much larger number. This is the same concept except that we are acting in the binary world rather than in the decimal world. The problem occurs when we cast a variable that has a value larger than what the new variable is capable of storing. So, if we go back to our decimal example, what happens if I want to store the value 199 in a variable that can only store two digits? The explanation of what happens on your PC is beyond what I want to get into in this course. Simply be aware of this issue and avoid it.

Important: It is important to know that arithmetic performed on a single type will remain an this regardless of the type of the variable it is assigned to. Consider the following:

```
int x = 1;
int y = 2;
float z;
z = x/y;
```

The above will result in the value 0.0 being assigned to `z`. This is due to the fact that `x/y` is performed over integers and the result is therefore stored as an integer and is then cast directly before the assignment. In order to force the arithmetic to take place on floating point values, one or more of the variables must be cast to a `float`. The resulting assignment should look like this:

```
z = (float) x/z;
```

4.7. The `void` Pointer

We discussed the `void` keyword very briefly in the context of a function's return value in Chapter 3. I mentioned that this keyword simply meant the function had no value to return. In the context of pointers, the `void` keyword allows you to declare a pointer to an unknown type (`void*`). This may be helpful in situations where you do not immediately know the type of the variable a pointer points to (or don't need to). I mention this as it is a construct that you will see in the next chapter, but we won't spend too much time with void pointers. Simply know that you can cast pointers in the same way you can cast other variables and you will simply need to cast a void pointer to a usable pointer before you use it. We will see an example of this in the next chapter.

5. Dynamic Memory

Until now we have statically allocated the memory for all our variables. That is, we assume that we know beforehand how much memory our data needs. For example, we declared all our arrays with constant values. However, we may have a problem in situations where we don't know how large the data will be at the time we are writing our code. Let us assume that we are tasked with writing a program that allows the user to repeatedly enter names into our program and it is our program's job to store them (in order to sort them alphabetically, for example). Well, we can't know whether the user will enter two names or 200 before hand. There is simply no way for us to know. We could simply declare a very large array and hope that a user never enters more than this. This is not a good solution, however. First, it unnecessarily wastes memory if the user only enters two names. Second, it creates an arbitrary maximum value that we would like to avoid.

In fact there is a way to handle this situation in which we can allocate memory at runtime. We call this dynamic allocation of memory.

5.1. Address Space Layout

This section will cover some points not directly related to C, but will rather give you some background as to what is happening on the machine. It will cover some computer architecture and operating systems (OSs) principles, though just enough to give you some background.

When we code in C and compile that code, we create a **program**. Once that program is running on the system, we call it a **process**. If the same program is running multiple times, we say the same program is running as multiple processes. In fact, there are many processes running on your system constantly. Through the magic of modern OSs, each one of these processes may run as if it has all the memory to itself. That is, it (or you, as the programmer) do not need to be concerned with sharing memory. You may act as if it all belongs to you.

In addition to this, we may assume that a process has the maximum amount of memory available to it regardless of how much physical memory is actually in the machine. On a 32-bit architecture, this maximum amount of possible memory is 4GB of memory and we can assume that we can use all of it, even if there is only 2GB of physical memory in the machine! You may ask why a 32-bit machine has a maximum memory amount of 4GB. This is simple. On a 32-bit machine, the largest number we can represent in binary is 11111111111111111111111111111111 (32 1s). This number in decimal form is 4,294,967,295 (4GB). That is, if you consider the memory to be an array of bytes as I mentioned in Chapter 4, the largest index the machine is capable

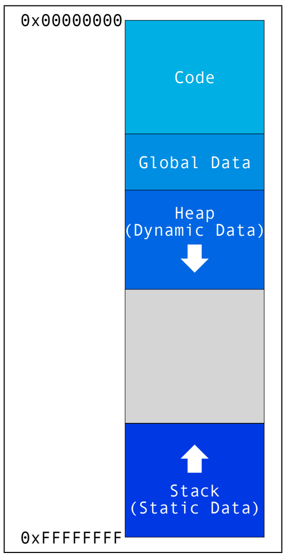


Figure 5.1.: A Process' Address Space

of using and understanding is 4,294,967,295. So even if there were more memory the machine would be unable to address it because its index is a number higher than what it can produce! On a 64-bit machine, this maximum number is obviously much higher, but the same principle applies.

In computer science we often use other numbering systems such as binary (as you have seen) and hexadecimal. Hexadecimal is a base-16 numbering system that is often used in computing. I mention this because 4,294,967,295 is often represented as 0xFFFFFFFF in hexadecimal and you should recognize this.

Now that we have covered this we can talk about a process' address space. This address space is simply the process' view of the memory it has and is depicted in Figure 5.1. In this figure, you see that the process' memory begins at 0x00000000 at the top of the figure and goes to 0xFFFFFFFF at the bottom. As we mentioned, the process may assume it has 4GB of memory and doesn't need to share it. We also see in the figure that this address space is split up into sections. The first from the top is the **code** section. This is where the program's instructions reside. All the code that you write in C is translated to something the machine can understand and is stored here. The **global data** section below that is where all global variables are stored. Below that we have the **heap**. The heap is where all the dynamic data we talked about in the previous section is stored. This section may grow as the program runs and as the process allocates additional memory. Finally, the the very end of the address space is the **stack**. This is where all the static variables in your various functions (including the `main` function) are stored. As additional functions are called this section may also grow. Since both the stack and the heap may grow at run time, they are placed at opposite ends of the remaining space and grow towards one another.

I mention this as it may be interesting to see where the dynamic data we will learn to allocate in the next section is stored and how that differs from where static data is stored.

5.2. `malloc` and `free`

In this section, we will look at the specific mechanisms for allocating and deallocating memory.

5.2.1. Allocation

In C, we allocate dynamic memory with the `malloc` function. This function is very straightforward and it accepts the size of the memory block you want to allocate in bytes while it returns a `void` pointer to the newly allocated memory. This function returns a `void` pointer because it does not know what you want to do with that memory, therefore it cannot associate a type with it. However, we learned in the previous chapter that we can cast `void` pointers to any other pointer so this is not an issue for us. For example, if we want to allocate an integer on a 32-bit system and we know that an integer is four bytes in length, we simply call:

```
void *void_pointer;
void_pointer = malloc(4);
```

After the above statement has executed, the variable `void_pointer` is a `void` pointer pointing to a newly allocated four bytes of memory. We can now cast `void_pointer` to an integer pointer and assign it to an integer pointer. We can also do this casting immediately as in the following example:

```
int *int_pointer;
int_pointer = (int*) malloc(4);
```

In the above example, we are casting the output of `malloc` directly to an integer pointer.

As a final point of interest, C offers the `sizeof` built-in function. This function accepts a type as input and outputs the size of that type for the current architecture. This allows us to write portable code and not to have to worry about how many bytes each type needs. Finally, an example pulling all of this together.

```c
#include <stdlib.h>
...
  int *x;
  float *y;

  x = (int*) malloc(sizeof(int));
  y = (float*) malloc(sizeof(float));
...
```

Listing 5.1: Memory allocation with `malloc`

In the above example, we see on Line 1 that `malloc` requires `stdlib.h` to be included. We then see the declaration of an integer and floating point pointer on Lines 3 and 4. Finally, on Lines 6 and 7, we see the `malloc` and `sizeof` functions in action.

5.2.2. Deallocation

As opposed to statically allocated memory, which is automatically deallocated when the function in which it's allocated returns, dynamically allocated memory remains allocated until you explicitly deallocate it. This is done with the `free` function and is equally straightforward to use. It simply accepts a `void` pointer to memory that was dynamically allocated with `malloc`. Consider the following example:

```
1 #include <stdlib.h>
2 ...
3   int *x;
4
5   x = (int*) malloc(sizeof(int));
6 ...
7   free((void*)x);
8 ...
```

Listing 5.2: Memory deallocation with `free`

You will notice, once again, that `free` requires the `stdlib.h` header to be included as shown on Line 1. `free` is actually used on Line 7 and hopefully you will agree that its use is very straightforward. It simply takes a `void` pointer to any previously allocated memory region.

It is very important that you only free memory that was explicitly allocated by you. If you attempt to free statically allocated memory, you will have issues. Equally important is that you **must** free all memory that you allocate. Generally, we say make sure you have one call to `free` for each call to `malloc`. This might not seem important to you now and you will not notice any issues in your assignments. It is however very important as the programs you write get larger (and is simply considered good coding practice). If you have ever heard the term **memory leak**, it refers to memory that is allocated, but never deallocated. This results in a process that constantly allocates more and more memory without freeing it. This will eventually lead to your process running out of memory.

6. Preprocessor Directives

In this chapter we will very briefly discuss a few useful preprocessor directives. In fact, we have been using one such directive since Chapter 1. This is the `#include` directive. This directive tells the compiler what additional header files to pull into the program when compiling.

Before we get into further detail however, I will explain exactly what a preprocessor directive is. These are essentially commands for the compiler and not for the machine that always start with '`#`'. That is, these commands are interpreted by the compiler and are not translated into machine code.

6.1. The `#include` Directive

As mentioned, this directive instructs the compiler to include other files in which external functions are declared. For example, you may have noticed throughout this primer that the use of various functions requires that a specific header file be included. This is due to the fact that the function you are interested in using is declared in that file. In order for the compiler to be able to compile your program it is necessary that it knows where all the functions you use are declared and defined. As you know the `#include` directive is generally called at the beginning of a file and takes the following form:

```
1 #include <stdlib.h>
2 #include ”custom.h”
3 ...
```

Listing 6.1: Using the `#include` directive

You will hopefully recognize the directive as it is used on Line 1. On Line 2 you might notice the file is enclosed in quotes as opposed to angled brackets as you are used to. This simply tells the compiler to look in the same directory as the current source file. This is used when you are working in larger projects that span multiple files. We will discuss this further in Chapter 7.

6.2. The `#define` Directive

This directive allows you to define a macro which is simply a fragment of code which has been given a name. Whenever this macro is used in the text of your program the compiler simply replaces it with whatever you have defined it as. Consider the following:

```
1 ...
2 #define PI 3.14
3 #define short_name() very_very_very_long_function_name()
4 ...
```

Listing 6.2: Using the `#define` directive

A `#define` directive differs from a normal variable definition in that it cannot be changed at runtime. You can simply think of the compiler going through your program and replacing all instances of `PI` with `3.14` before it starts compilation, for example. You see such a `#define` on Line 2 of the above listing. On Line 3 you see a long function name defined as a shorter one.

6.3. The `#ifdef`, `#ifndef`, `#else`, and `#endif` Directives

These directives form an "if" construct for the compiler. That is, we can tell the compiler to only bother compiling the following code if a certain macro has been defined. For example, we may want to make our code compilation work on various architectures that require different steps. Consider the following:

```
1 ...
2 #define X86 1
3 #ifdef X86
4 ...
5 #else
6 ...
7 #endif
8 ...
```

Listing 6.3: Using the `#ifdef` `#else` and `#endif` directives

In the above example, we see that `X86` is explicitly defined. Then on Line 3, the compiler will look to see if it is defined due to the `#ifdef` directive. Upon seeing that it is defined it will compile everything between the `#ifdef` and the `#else` directives and will ignore everything between the `#else` and the `#endif` directives.

This construct is also often used in header files to avoid multiple declarations of the same file as we will see in the next chapter.

7. Programming with Multiple Files

In this chapter we will explore using header files and working with larger projects that span multiple files. We will begin by considering header files.

Up to this point we have used several header files in that we included them into our project, however in this section we will look at creating our own. The concept is very simple. We generally create a header file for each C file we create in which the header file contains declarations that are usable by other files. At this point it may be worthwhile explaining that functions can be declared without being defined. That is, we can declare a function in the header file and define it in the C file. We declare a function in the following manner:
`int func1(char*, int);`
The above declaration tells us that func1 will be defined and that it will return an integer and accepts a character pointer and an integer. This is actually enough information for us to be able to use the function if we know what it does. How it is implemented might not even concern us. Such declaration can be put into a header file so that other files can use the functions without having to define them again. Consider the three files below:

```c
#ifndef ARITHMETIC_H
#define ARITHMETIC_H

int addition(int, int);
int subtraction(int, int);

#endif
```

Listing 7.1: arithmetic.h

```c
int addition(int x, int y){
  return (x+y);
}

int is_greater(int x, int y){
  if(x > y){
    return 1;
  }
  else{
    return 0;
  }
}

int pos_subtraction(int x, int y){
  if(is_greater(y,x)){
    return 0;
  }
  return (x-y);
}
```

Listing 7.2: arithmetic.c

```c
#include <stdio.h>
#include "arithmetic.h"

int main(int argc, char **argv){
  printf("The sum of 3 and 5 is %d\n",addition(3,5));

  return 0;
}
```

Listing 7.3: my_program.c

There are several things you should notice. First, notice in *arithmetic.h* that we use the `#define`, `#ifndef`, and `#endif` directives. This is very useful is large projects as the header file only needs to be included once. This construct makes sure that it is only ever included into a project once. You will notice it first checks whether the macro `ARITHMETIC_H` is defined and if it is defined it ignores the entire file. If it is not defined it immediately sets `ARITHMETIC_H` and performs the declarations. This way once it has been included once, it will be ignored on each subsequent include.

You will also notice the function declarations in *arithmetic.h* and notice that these functions are then implemented in *arithmetic.c*. You do not have to declare all the functions you define in *arithmetic.c* in *arithmetic.h*, however they will not be directly visible in another file that includes the header file. You will notice that the function `is_greater` is defined in *arithmetic.c*, but not declared in *arithmetic.h*. This simply means that any file that includes *arithmetic.h* will not be able to directly use this function. It is meant only as a function internal to *arithmetic.c*.

Additionally, you may notice that *arithmetic.c* does not have a `main` function. This is due to the fact that it is designed to be used as an auxiliary library for the project. We cannot have more than one `main` function per program. Therefore if *arithmetic.c* had a main function and *my_program.c* wanted to include its functions, we would have an issue.

Finally, you will notice that in *my_program.c* on Line 2, *arithmetic.h* is included and is enclosed in quotation marks rather than the angled brackets. This tells the compiler to look for *arithmetic.h* in the same directory as *my_program.c* as mentioned in Chapter 6.

This is, of course, a very simple example of a multi-file program, but hopefully it is enough to convey the idea.

8. Debugging

In this chapter, we will cover debuggers. Debuggers are often overlooked by newer programmers in favor of “printf debugging”. Perhaps this due to the fact that debuggers seem complex and daunting. In this chapter we will try to debunk this myth.

The first thing that needs to be mentioned is that it is possible to store debugging symbols within the binary that the compiler generates. These debugging symbols help the debugger map machine instructions back to the source code. Remember that generally the compiled binary is a translation of your source code and the actual source code is lost. That is, without these debugging symbols you would have to debug your program on a machine instruction level. In order to tell the compiler to include this information, we use `gcc` with the `-g` flag. For example:

```
$ gcc -g -o test test.c
```

This will generate a binary with the name, `test` and this binary will contain the debugging symbols. You will notice that the binary is also larger than if it were compiled without debugging symbols.

8.1. Valgrind - Memcheck

From their website[1], “Valgrind is an instrumentation framework for building dynamic analysis tools.”. That is, valgrind is actually much more than a simple debugger. However, there are tools that leverage valgrind which can be very helpful in debugging C programs. Specifically, in this course we will consider *Memcheck*. Memcheck is a tool for detecting memory leaks, double frees, accesses of unallocated or uninitialized memory, etc.

Installing Valgrind locally is generally as simple as searching through your Linux distribution’s application repository. To run it, simply call `valgrind` with the appropriate options and the binary you want to debug. If you are interested in using Memcheck, we call valgrind as follows (assuming our binary is called `test` and has debugging symbols included):

```
$ valgrind --tool=memcheck ./test
```

This will call and run your program. If your program requires input, you may interact with it as if you had run it natively. After your program has run, Memcheck will provide a summary of memory usage and errors. If you indeed have a memory leak, you may notice that Memcheck only provides a summary of the problem with the hint that we should “Rerun with –leak-check=full to see details of leaked memory”. This would look as follows:

```
$ valgrind --tool=memcheck --leak-check=full ./test
```

Remember, a memory leak is simply a situation in which memory is allocated and never specifically deallocated.

If you have a memory leak in your program, you may see a report that looks something like this:

```
5 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C2A2DB: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
   by 0x40054C: allocate_something (test.c:11)
   by 0x40056C: main (test.c:18)
```

The first line tells us that a single block of 5 bytes (i. e., some data structure of 5 bytes) was allocated and no pointer was maintained. The indented portion after the first line is a *call trace* or *stack trace*. This provides a sequence of functions that were called to get to the allocation within our code, starting at the bottom. We see (starting at the bottom), that the first function is the `main` function. You will notice the text in parenthesis after the name of the function. In this case, it is “test.c:18”. This indicates to us that, at this location, the program called the next function in our trace. In this case, it is the `allocate_something` function. Here we see that this function calls malloc on line 11 of *test.c* from the text in parenthesis and at the call to malloc the trace stops. This indicates to us that the offending allocation happens on line 11 of *test.c*, which in fact is the case. With this information, we can now try to determine the situation that lead to this allocation never being freed and correct it.

[1] `http://valgrind.org/`

8.1.1. Memcheck Error Overview

In addition to the memory leaks outlined above, Memcheck is able to detect other memory errors. If Memcheck finds other errors, it will print out a message along with a call trace similar to the example we saw above with respect to memory leaks. This section briefly describes each of these error types that Memcheck is capable of detecting. The following information is taken from the official Valgrind documentation[2]. For a more detailed explanation of each error, please see this reference.

Invalid reads/writes This error indicates that your program is reading from or writing to memory which it shouldn't.

Use of uninitialized values This error indicates that your program is using a variable which has not be initialized. That is, the variable has never been assigned a value.

Use of invalid values in system calls You will probably not see this error all to often unless you are using system calls directly. This error indicates that some value passed to a system call is uninitialized or unaddressable.

Illegal frees This error means your program is freeing an area of memory that cannot or should not be freed. This generally comes in the form of a *double free.* A double free simply means that your program tries to free the same area of memory twice.

Inappropriate allocation function As we are exclusively using `malloc` and `free` for now, this error should never occur. There are in fact more way to allocate memory in C++. This error means that your program is trying to "mix and match" allocation and deallocation functions. That is, if I use a C++ specific allocation mechanism, I cannot use `free` to deallocate it, but must rather use a specific deallocation function.

Overlapping source and destination This error indicates that you are calling a function such as `strncpy`, which requires a source and destination buffer, with overlapping source and destination buffers. This can result in strange results and should never happen.

Memory leak detection This error was explained in a bit further detail in the previous subsection. It indicates that memory has been allocated and was never deallocated (freed).

8.1.2. Reference

For a look at the complete Valgrid documentation, see `http://valgrind.org/docs/manual/`.

[2]`http://valgrind.org/docs/manual/mc-manual.html#mc-manual.errormsgs`

8.2. GDB

GDB stands for “GNU Debugger” and is the defacto standard debugger for all GNU operating systems. GDB is quite powerful and has many features. We will concentrate on the most basic features required for the debugging of simple programs.

To get started with GDB, simply run gdb with your binary as the first argument. Make sure that your binary was compiled to include debugging symbols. The call should look like this (assuming your binary is called *test*):

```
$ gdb ./test
```

This will start gdb and give you a command line environment. Notice that your program has not yet started execution at this point.

Also, you can exit GDB at any time with the `quit` or `q` command on the GDB command line.

8.2.1. Executing your Program

To execute your program once you have started GDB, use the `run` command. If you want to execute your program with some parameters, you may include these after the `run` command. For example, to start your program execution with the command line parameters *abc* and *123*, simply run:

```
(gdb) run abc 123
```

This will run your program with the command line arguments `abc 123`.

You will notice that your program will simply run through as if it had been executed on the command line. The exception to this is if your program ends in an unhandled exception (e. g., a segmentation fault). In this case you will be able to inspect the contents of variables and registers immediately after the exception. This can be helpful if you are trying to debug segmentation faults, for example. Additionally, you may press `CTRL+C` at any time while your program is running it to pause execution. This will pause execution, but you cannot be sure exactly where your program will pause. Finally, for more fine grained control, your program will pause execution if it hits a breakpoint that you set at an earlier time point. Breakpoints and their use will be discussed in a further subsection.

There are some additional mechanisms by which GDB can pause the execution of your program, but we will not cover them in this primer.

8.2.2. Listing Source Code

At any time, you can have DBG list the source code around your current position with the `list` command. If you would like to print the source code for a specific location, you can specify a filename and line number in the form of `filename:line_number`. For example,

```
(gdb) list test.c:7
```

will print lines of source code around line 7 of *test.c* (assuming test.c exists), while a simple

```
(gdb) list
```
will print lines of source code around your current location.

8.2.3. Breakpoints

As mentioned in a previous subsection, you may set breakpoints in your code. A breakpoint is simply a point in the code where you would like the execution to pause so that you can further inspect the state of your program. This might include looking at the values of variables or registers, for example.

Setting Breakpoints

To set a breakpoint use the `breakpoint` command (or `b` for shorthand).

The `breakpoint` command requires some additional input. GDB must know where you want to set your breakpoint. There are two variations that we will cover in this lecture.

First you can set a breakpoint at the beginning of any function by passing the function name to the `breakpoint` command. For example,
```
(gdb) breakpoint func1
```
will set a breakpoint at the start of the `func1` function (given this function exists).

Additionally, you can set a breakpoint by passing the filename and line number to GDB. To do this, you simply pass this information to the `breakpoint` command in the form of `filename:line_number`. For example,
```
(gdb) breakpoint main.c:12
```
will set a breakpoint on line 12 of the *main.c* file (given this file exists).

Once you have set your breakpoints, you can run your program as described in a previous subsection and GDB will automatically pause your program when a breakpoint is hit.

Listing Breakpoints

Of course, you can set multiple breakpoints at once. To list all your breakpoints, use:
```
(gdb) info break
```
This will list all set breakpoints.

Deleting Breakpoints

To delete a breakpoint use the `delete` command. By itself, this will delete all breakpoints. If you pass it an argument, you can specify which breakpoint to delete. You must pass `delete` the breakpoint number of the breakpoint you wish to delete as given by the `info break` command. For example, if I wish to delete breakpoint 2 as specified by `info break`, you would execute:
```
(gdb) delete 2
```
or to delete all breakpoints, you would execute:
```
(gdb) delete
```

8.2.4. Continuing Execution

Once your program execution is paused (usually through the use of a breakpoint), you will eventually want to continue execution. There are several ways to go about this.

Continue

In the simplest case, you may simply want to continue execution until the next breakpoint is hit or your program ends. In this case, you can simply execute the command `continue` (or `c` for shorthand). That is, to simply continue execution, execute:

```
(gdb) continue
```

Finish

Another simple case is the situation in which you continue execution until the end of the current function. This can be accomplished with the `finish` command. Its use is straightforward, simply execute:

```
(gdb) finish
```

Single-stepping

In addition to simply continuing execution of your program you also have the ability to walk though your program by executing one line at a time. This is called *single-stepping.* There are two commands that can perform this for you, namely `step` (or `s` for shorthand) and `next` (or `n` for shorthand). They differ in one very important respect: `step` will single-step **into** functions, while `next` will single-step **over** functions. That is, `next` will execute a function as if it is a single line and `step` will take you into the function that is being called.

These are very simple commands that do not require any parameters for the purposes of this primer and can be used as follows:

```
(gdb) step
```

and

```
(gdb) next
```

8.2.5. Inspecting State

Any time your system is paused you can also inspect the current state of the machine. This includes inspecting the values of variables and registers.

Show Arguments

At any time while the system is paused, you can inspect the arguments of the current function. This is accomplished with the `info args` command. That is,

```
(gdb) info args
```

will print all arguments of the current function along with their current values.

Show Local Variables

In addition to the arguments of a function, GDB will allow you to inspect the local variables of the current function. This can be achieved with the `info locals` command. For example,

```
(gdb) info locals
```

will print all local variables and their current variables within the current function.

Show Arbitrary Variables

Finally, it also possible to print the value of arbitrary variables with the `print` command. This command must be passed the name of a variable as an argument. Of course, any printed variable must currently be in scope. This command may be useful if you are interested in a global variable or a variable that is dynamically allocated. For example,

```
(gdb) print x
```

will print the value of a variable named x (given a variable of this name exists in the current scope).

8.2.6. References

As I mentioned at the beginning of this section, GDB has many functions above what I have introduced here. There are many additional features that you may find useful as you get comfortable with using debuggers. For the complete online documentation, see `http://sourceware.org/gdb/current/onlinedocs/gdb/`.

Additionally, there are several nice GDB "cheatsheets" out there with a reference to the most commonly used commands. Two that might find useful can be found at `http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf` and `http://www.cs.berkeley.edu/~mavam/teaching/cs161-sp11/gdb-refcard.pdf`.

A. ASCII Table

Hex	Dec	Char	Hex	Dec	Char
0x00	000	*null*	0x40	064	`@`
0x01	001	*start of heading*	0x41	065	`A`
0x02	002	*start of text*	0x42	066	`B`
0x03	003	*end of text*	0x43	067	`C`
0x04	004	*end of transmission*	0x44	068	`D`
0x05	005	*enquiry*	0x45	069	`E`
0x06	006	*acknowledge*	0x46	070	`F`
0x07	007	*bell*	0x47	071	`G`
0x08	008	*backspace*	0x48	072	`H`
0x09	009	*horizontal tab*	0x49	073	`I`
0x0a	010	*new line*	0x4a	074	`J`
0x0b	011	*vertical tab*	0x4b	075	`K`
0x0c	012	*new page*	0x4c	076	`L`
0x0d	013	*carriage return*	0x4d	077	`M`
0x0e	014	*shift out*	0x4e	078	`N`
0x0f	015	*shift in*	0x4f	079	`O`
0x10	016	*data link escape*	0x50	080	`P`
0x11	017	*device control 1*	0x51	081	`Q`
0x12	018	*device control 2*	0x52	082	`R`
0x13	019	*device control 3*	0x53	083	`S`
0x14	020	*device control 4*	0x54	084	`T`
0x15	021	*negative acknowledge*	0x55	085	`U`
0x16	022	*synchronous idle*	0x56	086	`V`
0x17	023	*end of trans. block*	0x57	087	`W`
0x18	024	*cancel*	0x58	088	`X`
0x19	025	*end of medium*	0x59	089	`Y`
0x1a	026	*substitute*	0x5a	090	`Z`
0x1b	027	*escape*	0x5b	091	`[`
0x1c	028	*file separator*	0x5c	092	\
0x1d	029	*group separator*	0x5d	093	`]`
0x1e	030	*record separator*	0x5e	094	`^`
0x1f	031	*unit separator*	0x5f	095	`_`

Hex	Dec	Char	Hex	Dec	Char
0x20	032	*space*	0x60	096	‘
0x21	033	!	0x61	097	a
0x22	034	"	0x62	098	b
0x23	035	#	0x63	099	c
0x24	036	$	0x64	100	d
0x25	037	%	0x65	101	e
0x26	038	&	0x66	102	f
0x27	039	’	0x67	103	g
0x28	040	(	0x68	104	h
0x29	041	)	0x69	105	i
0x2a	042	*	0x6a	106	j
0x2b	043	+	0x6b	107	k
0x2c	044	,	0x6c	108	l
0x2d	045	-	0x6d	109	m
0x2e	046	.	0x6e	110	n
0x2f	047	/	0x6f	111	o
0x30	048	0	0x70	112	p
0x31	049	1	0x71	113	q
0x32	050	2	0x72	114	r
0x33	051	3	0x73	115	s
0x34	052	4	0x74	116	t
0x35	053	5	0x75	117	u
0x36	054	6	0x76	118	v
0x37	055	7	0x77	119	w
0x38	056	8	0x78	120	x
0x39	057	9	0x79	121	y
0x3a	058	:	0x7a	122	z
0x3b	059	;	0x7b	123	{
0x3c	060	<	0x7c	124	|
0x3d	061	=	0x7d	125	}
0x3e	062	>	0x7e	126	˜
0x3f	063	?	0x7f	127	*delete*