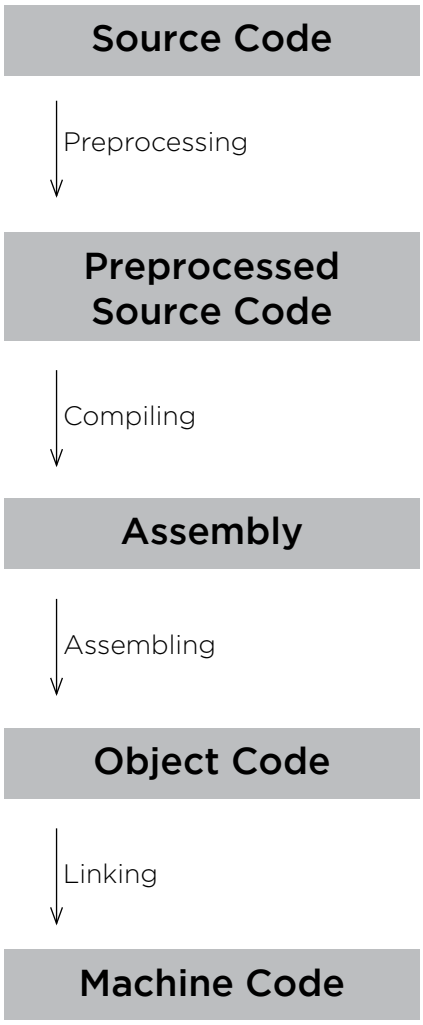
CS50 Compiling

Overview

Compiling is the process of translating source code, which is the code that you write in a programming language like C, and translating it into **machine code**: the sequence of 0s and 1s that a computer's central processing unit (CPU) can understand as instructions for how to execute the program. Although the command `make` is used to compile code, `make` itself is not a compiler. Instead, `make` calls upon the underlying compiler `clang` in order to compile C source code into object code.

Key Terms

- compiling
- machine code
- preprocessing
- assembly
- object code
- linking



Preprocessing

The entire compilation process can be broken down into four steps. The first step is **preprocessing**, performed by a program called the preprocessor. Any source code in C that begins with a `#` is a signal to the preprocessor to perform some action.

For example, `#include` tells the preprocessor to literally include the contents of a different file in the preprocessed file. When a program includes a line like `#include <stdio.h>` in the source code, the preprocessor generates a new file (still in C, and still considered source code), but with the `#include` line replaced by the entire contents of `stdio.h`.

Compiling

After the preprocessor produces preprocessed source code, the next step is to compile (using a program called a compiler) C code into a lower-level programming language known as **assembly**.

Assembly has far fewer different types of operations than C does, but by using them in conjunction, can still perform the same tasks that C can. By translating C code into assembly code, the compiler takes a program and brings it much closer to a language that a computer can actually understand. The term "compiling" can refer to the entire process of translating source code to object code, but it can also be used to refer to this specific step of the compilation process.

Assembling

Once source code has been translated into assembly code, the next step is to turn the assembly code into object code. This translation is done with a program called the assembler.

Object code is essentially machine code with some non-machine code symbols. If there's only one file that needs to be compiled from source code to machine code, the compilation process is over now. However, if there are multiple files to be compiled, a file's object code only represents part of the program and an additional step is required. The object code file's non-machine code symbols denote how the file fits with the other parts of the program. The entire program is put together in a process called linking.

Linking

If a program has multiple files that need to be combined into a single machine code file (such as if a program includes multiple files or libraries like `math.h` or `cs50.h`), then one final step is required in the compilation process: **linking**. The linker takes multiple different object code files, and combines them into a single machine code file that can be executed. For example, linking the CS50 Library during compilation is how the resulting object code knows how to execute functions like `get_int()` or `get_string()`. It is important to note that only one file can have a `main` function so that the program knows where to begin.

CS50 Bugs and Debugging

Overview

A **bug** is an error in code which results in a program either failing, or exhibiting a behavior that is different from what the programmer expects. Bugs can be frustrating to deal with, but every programmer encounters them. **Debugging** is the process of trying to identify and fix bugs that exist in code. Programmers will often do this by making use of a program called a **debugger**, which assists in the debugging process.

Key Terms

- bugs
- debugging
- debugger
- breakpoint
- `debug50`

Debugging Basics

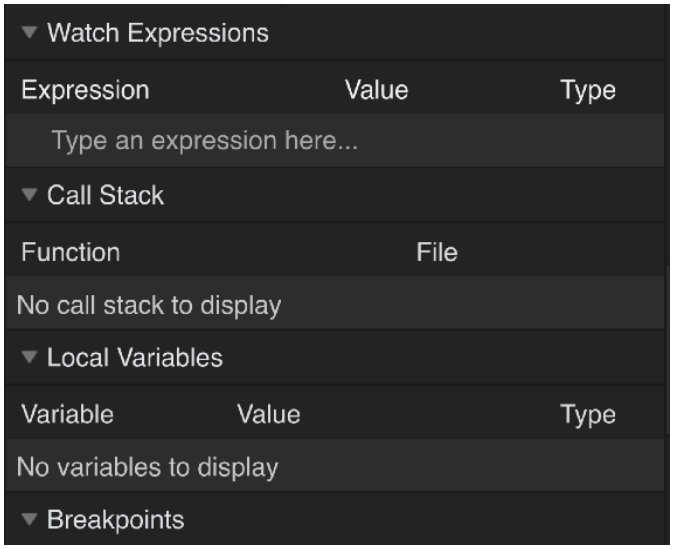
Programs generally perform computations much faster than a human possibly could, which makes it difficult to see what's wrong with a program just by running it all the way through. Debuggers are valuable because they allow a programmer to freeze a program at a particular line, known as a **breakpoint**, so that the programmer can see what's happening at that point in time. It also allows the programmer to execute the program one line at a time, so that the programmer can follow along with every decision that a program makes.

Using `debug50`

`debug50` is a program that we've created that runs a built-in graphical debugger in the CS50 IDE. Before you run `debug50`, you must set at least one breakpoint. If you have a general sense for where your program seems to be going wrong, it may be wise to set a breakpoint a few lines before there, so that you can see what's happening in your program as it moves into the section that you believe is causing the problem. If you're not sure at all, then it's totally fine to set a breakpoint at the first line of your `main` function so that you can step through the entire code from the beginning. To set a breakpoint, click the space to the left of the line number of your program. You will see a red dot appear in that space and it will be added to the list of all your breakpoints at the bottom of the graphical debugger window. You can remove a breakpoint by clicking on the red dot next to the line number.

Once you're satisfied with your breakpoints, run your program with as usual with `debug50 ./program_name` and any command-line arguments. Your program will run, and automatically stop at any breakpoints. In the debugger window, you'll see several tabs you can interact with.

At the top of the window, you'll see five buttons. The first, the blue triangle, will run your code until it hits the next breakpoint. The next button, the curved arrow, allows you to skip over a block of code. Next to that is the down arrow which allows to move through your code slowly one line at a time. The last arrow allows you to step out of a function (other than `main`). The last icon, the outline of a circle, clears all of the breakpoints that you set in your program.



The window at left summarizes some some features of `debug50`. In the "Watch Expressions" tab, you can type in a variable or function you want to watch while your code runs.

The "Call Stack" tab displays what function the line of code you are on is in and the file path for the programming that is running.

The "Local Variables" and "Breakpoints" are pretty intuitive. For each variable, you can see it's name, value and type. You can even override variable values by clicking on the value and typing in a new value. For each breakpoint you've marked, you'll see the file name, line number, and what appears on that line of code. You can turn off breakpoints from this tab by clicking on the checkbox next to the breakpoint, or delete it by clicking on the 'x' in the top right corner of the breakpoint when you hover over it.

CS50 Arrays and Strings

Overview

Recall that variables are used to store values. Quite frequently, we may want to use multiple variables to store a sequence of values: like a sequence of 10 test scores, or 50 addresses. For situations like these, C has a data structure called an **array**: which stores multiple values of the same type of data. For instance, an array of `int`s would store multiple `int` values back-to-back. The **string** type that you have been using is really just an `array` of `char`s.

Key Terms

- array
- string
- size
- index
- null-terminator

```
1  int ages[5];
```

0	1	2	3	4

```
2  ages[0] = 28;
```

0	1	2	3	4
28				

```
3  ages[1] = 15;
4  ages[2] = ages[1];
5  ages[3] = ages[1] - 1;
6  ages[4] = 17;
```

0	1	2	3	4
28	15	15	14	17

```
7  for (int i = 0; i < 5; i++)
8  {
9      ages[i] += 1;
10 }
```

0	1	2	3	4
29	16	16	15	18

Arrays

Like variables, arrays are declared by first stating the type of the data to be stored, followed by the name of the array. In brackets after the name of the array is the **size** of the array: which defines how many values the array will hold. For example, line 1 at left declares an array of 5 `int`s.

You can visualize an array as a sequence of boxes, each one holding a value, and each one with a numbered **index**, which is a number that can be used to access a specific value in an array. In C, arrays are zero-indexed, meaning that the first item in an array has index `0`, the second item has index `1`, etc.

To access a particular value in an array, use the name of the array, followed by the desired index in brackets. Line 2 at left sets the value of the first item in the `ages` array (the one at index `0`) to 28.

The value at each array index can be treated like a normal variable. For example, you can change its value, apply arithmetic or assignment operators to it.

Since each value in an array is referenced by its index number, it's easy to loop through an array. Lines 7 through 10 define up a `for` loop, which iterates through the entire array, and increases each age value by 1.

Strings

In C, a `string` is represented as an array of `char` values. Thus, when we write a line like `string s = "CS50";`, this information is stored as an array of `char`s, with one character at each index. The final index of a string in C is the **null-terminator**, represented by `'\0'`. The null-terminator is the character that tells a `string` that the `string` is over, and that there are no more characters in the `string`.

```
string s = "CS50";
```

0	1	2	3	4
'C'	'S'	'5'	'0'	'\0'

Since a `string` is just an array, you can index into the `string` just like you would index into any other array in order to access the value of a particular character. For instance, in the example above, indexing into `s[0]` would give you the character `'C'`, the first character in the string `"CS50"`.

This also makes it very easy to use a loop to interate through a string and perform computation on each individual character within a string, by first initializing the loop counter to 0, and repeating until the last index of the string. The function `strlen()` takes in a string as input, and returns the length of the string as an integer, which may help in determining how many times the loop should repeat.

CS50 Command-Line Interaction

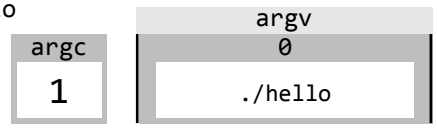
Overview

When running a program from the command line, you've generally executed a command like `./program_name` at the command line. C also allows you to specify a program's **command-line arguments**, which allows the person running the program to pass arguments into the `main` function of the program by specifying the arguments at the command line. This offers an alternative means of providing input to a program beyond just requesting input while a program is running, such as with `get_string()`.

Key Terms

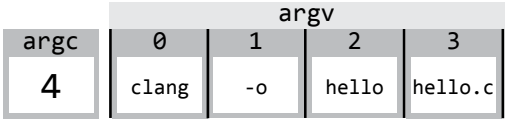
- command-line arguments
- argument count
- argument vector

`./hello`



`mkdir src`

`clang -o hello hello.c`



argc, argv

Many of the command-line programs that you have likely called before (`make`, `cd`, `clang`, `mkdir`) all take command-line arguments. In C, command-line arguments are passed into the main function as inputs. However, we've previously written our `main` functions to take no arguments (`void`).

To accept command-line arguments, we can revise the `main` function to take two arguments: `argc`, an integer, and `argv`, an array of `string`s.

`argc`, which stands for "**argument count**", represents the number of arguments passed into through the command line. Each word (separated by spaces) counts as its own argument, and the calling of the program itself (e.g. `./hello`) counts as an argument.

`argv`, which stands for "**argument vector**", is the actual array representing the arguments themselves. Each value in the array is a `string`.

If you were to look at `argc` and `argv` when calling a program with no arguments, like calling `./hello`, `argc` would be 1 (because the calling of the program is the only argument). `argv`, on the other hand, would be an array consisting of just one element: the string `"./hello"` stored at index `0`.

If you were to look at `argc` and `argv` when calling a program that does have arguments, like calling `mkdir src`, `argc` would be 2, since two arguments are passed in via the command line, and `argv` would be an array with two elements: the string `"mkdir"` stored at index `0`, and the string `"src"` stored at index `1`.

Using Command Line Arguments

Shown to the right is an example of a program which accepts command-line arguments. Notice on line 4 that the definition of the `main` function has changed to include the arguments `argc` and `argv`. No size of `argv` is specified on line 4, so that any array, regardless of its size, can be passed into the `main` function.

Inside of `main` function, the program loops through the array, starting at index `0`, and incrementing so long as `i < argc`. It's important to stop there, because the largest index of `argv` that you can access is `argc - 1` (since arrays are zero-indexed). During each iteration, the program prints out the value of `argv` at index `i`.

The result of the program is that each of the program's command line arguments is printed on a new line.

```
1  #include <cs50.h>
2  #include <stdio.h>
3
4  int main(int argc, string argv[])
5  {
6      for (int i = 0; i < argc; i++)
7      {
8          printf("%s\n", argv[i]);
9      }
10 }
```

CS50 Typecasting

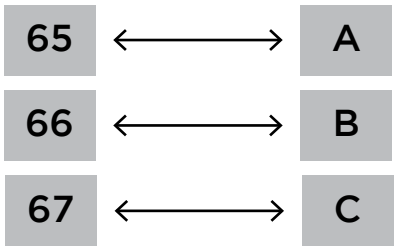
Overview

Recall that C has several different data types, including `int`s, `float`s, and `char`s. It may sometimes be necessary to convert variables from one data type to another data type. C allows us to do this via **typecasting** (or just "casting"). Typecasting allows you to cast data from one type to another type which is equally or less precise, but you cannot cast data from a type that is less precise to a type that is more precise.

Key Terms

- typecasting
- explicit typecasting
- implicit typecasting

```
1 | #include <stdio.h>
2 |
3 | int main(void)
4 | {
5 |     int x = 65;
6 |     printf("%i\n", x); // 65
7 |     printf("%c\n", (char) x); // A
6 | }
```



Chars and Ints

The ASCII standard, as you may recall, gives every letter a unique number to identify it: capital A is represented by the number 65, capital B by 66, and so on. Using typecasting, we can convert between integer values and `char` values.

Say, for instance, that we assigned an integer variable `x` to hold the value `65`. If we were to print the variable out on the screen (like on line 6 of the code to the left), then it would display the number `65` to the console.

On line 7, however, we've included a placeholder for a `char` instead of an `int` (as denoted by the `%c` symbol). We're still passing in `x` as an argument, but the code first casts `x` into a `char`. This is done by writing `(char)` in parentheses before the name of the variable. Placing a new type name in front of an existing variable to evaluate the variable as a different type is called **explicitly typecasting**: we are directly providing instructions to convert types.

While explicit typecasting in this situation is good practice from a style perspective (so that people reading your code can better understand what's happening), it's not actually necessary. If we were to exclude the `(char)` symbol from before the `x` in line 7 of the above code, the code will still print out the letter `A` (the ASCII mapping of the value `65`). Since we've included a placeholder for a `char`, the compiler is expecting a `char` to be passed in. If we pass an `int` in, the compiler will automatically try to interpret the value as a `char` instead. This is called **implicit typecasting**.

Ints and Floats

Typecasting is also valuable for converting between floating-point numbers and integers. Take the example at right. On line 6, we might want `b` to store the value of 28 divided by 5, which is 2.4. But line 6 actually sets `b` to be `2.0`. This is because the compiler sees a division between two `int`s, and thus presents the answer as an `int`, even though we're storing the value inside of a `float`. To get around this, we can first explicitly cast `a` to be a float, and then perform the division, as is done on line 7. In this case, `c` now correctly equals `2.4`.

Implicit typecasting can also be valuable when dealing with `int`s and `float`s. Since `int`s do not store digits past the decimal point, typecasting a `float` to an `int` is an easy way to truncate a number into an integer. On line 10 to the right, when we try to assign an `int` to be the floating-point value `d`, `d` is implicitly cast to be an `int`, getting rid of everything after the decimal point. The value of `e` is now `28`.

```
1  | #include <stdio.h>
2  |
3  | int main(void)
4  | {
5  |     int a = 28;
6  |     float b = a / 5;
7  |     float c = (float) a / 5;
8  |
9  |     float d = 28.523;
10 |     int e = d;
11 | }
```

CS50 Exit Codes

Overview

You may have noticed that the `main` function definition returns an `int`, but in the past we haven't been returning any value at the end of the main function. By default, if no return value is specified in the `main` function, the compiler will automatically assume that the `main` function returns `0`. The value that the `main` function returns is referred to as the program's **exit code**. As your programs become longer and more complicated, exit codes can be a valuable tool.

Key Terms

- exit code
- input validation

Using Exit Codes

By convention, if a program completed successfully without any problems, then it should return with an exit code of `0`. That's why the compiler assumes that if no return statement is provided at the end of `main`, the program should return `0`. You could, however, explicitly specify `return 0` at the end of a program.

Any non-zero exit code (commonly `1` or `-1`) conventionally means that there was some sort of error during the program's executing that prevented the program from completing successfully.

One common use of exit codes is during **input validation**: when the program checks to make sure that the inputs provided by the user are valid. For instance, if a program expects two command line arguments, but only receives one, it might return a non-zero exit code to signal an error.

```
 1  #include <cs50.h>
 2  #include <stdio.h>
 3
 4  int main(int argc, string argv[])
 5  {
 6      if (argc == 2)
 7      {
 8          printf("hello, %s\n", argv[1]);
 9          return 0;
10      }
11      else
12      {
13          return 1;
14      }
15  }
```

Take the above program, which takes (in addition to the program's name) a command line argument specifying the user's name. The program then says hello to the user.

Upon starting the `main` function, the program checks to see whether `argc` is 2. If it is, then the user's input is valid: the program can say hello, and successfully return with exit code `0`.

On the other hand, if the user didn't provide the correct input values (say, by not providing enough command line arguments, or by providing too many), then `argc` would not be equal to `2`.

The program would then execute the `else` block, which returns the number `1` as an exit code, indicating that there was an error in the program's execution.

Debugging

If you run programs normally from the command line, you won't actually see the return values that the `main` function returns. However, many debugging tools, which are programs designed for helping programmers find sources of problems in their code, will allow you to see the exit code with which the `main` function exited.

Knowing the exit code can be a valuable tool for determining why a program failed during the debugging process. In larger programs, which may include many instances of error checking and input validation, the program may return a different exit code for each error.

If the program fails, knowing which status code the program exited with can help determine what went wrong.