# Types, Arithmetic & Control

$C^5$ - C Crash Course Section 1

# In This Session

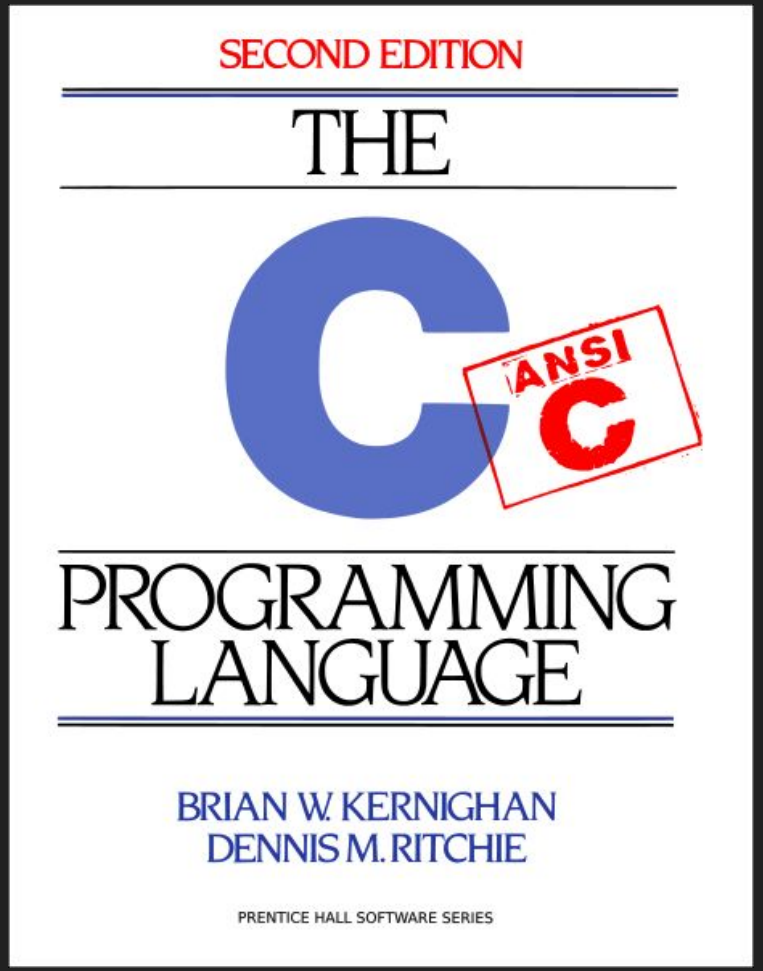
- What is C?
- A simple C program
- The `main` function
- Arithmetic in C
- Compiling
- Types
- **`If`**...**`else`**
- Displaying text
- User input

# What is C?

Created by Dennis Ritchie, it was a programming language that could be easily translated into *assembly* which could then be compiled into *machine language*.

It offers minimal support for structuring code and data and provides the most basic tools to create well structured programs.

It was one of the first compiled languages, allowing low level software (such as operating system kernels) to be written in high level code.



# Assembly and Machine Code

Computer processors take *instructions* that are a series of numerical values, each value can specify a specific instruction or what to data to perform the instruction on or with. This is machine code.

In order to write and read machine code relatively easily, *assembly* was developed. This uses words to represent the different aspects of the machine code.

```
Machine Code              Assembly

55                        push   rbp
48 89 e5                  mov    rbp,rsp
89 7d fc                  mov    DWORD PTR [rbp-0x4],edi
89 75 f8                  mov    DWORD PTR [rbp-0x8],esi
c7 45 f4 0c 00 00 00      mov    DWORD PTR [rbp-0xc],0xc
8b 75 f4                  mov    esi,DWORD PTR [rbp-0xc]
03 75 fc                  add    esi,DWORD PTR [rbp-0x4]
89 75 f4                  mov    DWORD PTR [rbp-0xc],esi
8b 75 f4                  mov    esi,DWORD PTR [rbp-0xc]
2b 75 f8                  sub    esi,DWORD PTR [rbp-0x8]
89 75 f4                  mov    DWORD PTR [rbp-0xc],esi
8b 45 f4                  mov    eax,DWORD PTR [rbp-0xc]
5d                        pop    rbp
c3                        ret
0f 1f 84 00 00 00 00      nop    DWORD PTR [rax+rax*1+0x0]
00
```

# What is C For?

The C language lets us write more human-readable language which is then *compiled* into the machine code that the computer understands. This lets code become more readily understandable and less prone to error.

The C code above gets *compiled* into the machine code below.

```c
int example (int a, int b) {
    int c = 12;
    c = c + a;
    c = c - b;
    return c;
}
```

```
Machine Code              Assembly

55                        push   rbp
48 89 e5                  mov    rbp,rsp
89 7d fc                  mov    DWORD PTR [rbp-0x4],edi
89 75 f8                  mov    DWORD PTR [rbp-0x8],esi
c7 45 f4 0c 00 00 00      mov    DWORD PTR [rbp-0xc],0xc
8b 75 f4                  mov    esi,DWORD PTR [rbp-0xc]
03 75 fc                  add    esi,DWORD PTR [rbp-0x4]
89 75 f4                  mov    DWORD PTR [rbp-0xc],esi
8b 75 f4                  mov    esi,DWORD PTR [rbp-0xc]
2b 75 f8                  sub    esi,DWORD PTR [rbp-0x8]
89 75 f4                  mov    DWORD PTR [rbp-0xc],esi
8b 45 f4                  mov    eax,DWORD PTR [rbp-0xc]
5d                        pop    rbp
c3                        ret
0f 1f 84 00 00 00 00      nop    DWORD PTR [rax+rax*1+0x0]
00
```

# What is a C Program

A C program consists of many parts. To begin with, we'll keep all the parts we make in one file.

- Comments - text for a human to read, to to be run as code
- References to other files - to use functions and other things defined in other files
- Constants - which let us give names to recurring values in our code
- Functions - which contain the code we want the computer to run

# Comments

These can appear anywhere in a C file and there are two types.

The first type uses a `//` to identify the comment. Everything in a line of text after the `//` is considered a comment.

The second type uses `/*` to mark the start of a comment and `*/` to mark the end. These allow comments to span multiple lines.

```c
int counter; // This number will count things

/* This function will count uppercase letters */
int countUppercase(char *text);

/*
This function will count lowercase characters
 */
int countLowercase(char *text);

/*
 * This is a bit easier to read.
 */
```

# Referencing Other Files

So that we don't need to rewrite code that appears in multiple files, we can reference other files using one of two methods. Both simply work by replacing themselves with the code in the file they reference when you compile.

The first method lets you reference files from the system, installed elsewhere or part of the *standard library*. This uses `<` and `>`

The second lets you reference files you provide yourself. This uses `"`.

These must appear *outside* a function.

```c
/*
 * This references the part of the _standard library_ for
 * _strings_, which lets our c file know that functions
 * such as:
 *   printf - write data to output
 *   scanf - read data from output
 *   strcmp - compare two text strings
 * exist as well as how they work.
 */
#include <string.h>

/*
 * This references our own file. This file contains
 * descriptions of how our own functions that have code
 * elsewhere work.
 *
 * It is assumed that there exists a file called `ourFile.h`
 * in the same directory as the file this appears in.
 */
#include "ourFile.h"
```

# Functions

Functions separate our code up into easily re-usable blocks, each given a unique name we can reference it by.

Every program must have one function called `main`. This is the function that is called when the program starts.

This function ends when it sees a `return` statement, and “returns” the value in the `return` statement which, for main, is always of type `int`.

We’ll look more at functions later.

```c
/*
 * This is the _entry point_ to our program, it is the function
 * that gets called first. It can appear in a few different ways,
 * but usually one of the following.
 */

int main (void) {
    // some code here...
    return 0;
}

int main (int argc, char *argv[]) {
    // some code here...
    return 0;
}

int main (int argc, char **argv) {
    // some code here...
    return 0;
}
```

# Types and Variables

A variable is single unit in the computer's memory where we can store data. These units must be given a name, so that we can identify which we are talking about, and a *type*, so we can identify how much memory they have to store data in and what they can be used for.

You must *declare* a variable before using it. This tells the code what *type* the variable is.

Names of variables can must start with a letter and may contain letters, numbers and underscores (_), and must not be keywords

```
/*
 * These are the most basic types and all store
 * a single number value, but each using varying size.
 */

// This variable is of the smallest type `char`
char tinyNumber;
// This is of the next smallest type `short`
short smallNumber;
// This is the most often used and is of the
// next smallest type `int`
int normalNumber;
// can only have positive values
unsigned int normalNumber;
// This is of the next smallest size `long int`
long int bigNumber;
// This is the largest size `long long int`
long long int biggestNumber;

/*
 * These are the standard types for storing fractional or
 * decimal values
 */
float decimalNumber;
double biggerDecimal; // twice as much memory as `float`
```

# Literals

We can also include explicit values in our code, called *literals*. Rather than referring to whatever data is stored in a variable, we can specify a specific value to use in code.

Variables can be single number values or *strings* of text.

```
/*
 * These lines uses the _literal_ value of `64` to set what
 * data is stored in our variable.
 */
someNumber = 64; // 64 as a decimal number (base 10)
someNumber = 0100; // 64 as an octal number (base 8)
someNumber = 0x40; // 64 as a hexadecimal number (base 16)

/*
 * Each character also has a numeric value, it's ASCII
 * value. You can get this by using `'` around just that
 * character.
 */
someNumber = '@' // The character @ has the value 64

/*
 * This line uses the _literal_ value of `72.25`
 */
someReal = 72.25;

/*
 * This line uses the _literal_ text `This is text` and sends
 * it to the output. Notice the use of `"` and not `'`.
 */
printf("This is text");
```

# Compiling with **`clang`**

Compiling with **`clang`** is pretty easy.

We use the option **`-Wall`** to give most possible warnings.

We use the option **`-Werror`** to treat all warnings as errors

We use **`-o progName`** to tell it what we want the program file to be called

We then list the files to compile.

```
host:crashCourse$ clang -Wall -Werror -o myProgram myProgram.c
host:crashCourse$ ./myProgram
Hello, World!
host:crashCourse$
```

# Simple Operations

The simplest operations we can perform on data are the basic mathematical operations, assigning data to a variable, and *boolean* operations (give a value of either *true* or *false*).

These must appear *inside* a function.

```
// Assignment operator is used to assign data to a variable
variableA = 2;
variableB = 3;
variableC = 4;

// We can use the standard mathematical operations
variableD = variableB + 5; // addition, sets variableD to 8
variableD = variableA - 4; // subtraction, sets variableD to -2
variableD = variableB * variableC; // multiplicaiton, variableD
becomes 12
variableD = variableD / variableA; // division, variableD becomes
6

// Modify the existing value in a variable
variableD = 4; // variableD becomes 4
variableD += 1; // variableD becomes 6
variableD -= 3; // variableD becomes 3
variableD *= 8; // variableD becomes 24
varibaleD /= 2; // variableD becomes 12

// Increment and decrement (always by 1)
variableD = 4; //variableD becomes 4
variableD++; //variableD becomes 5
variableD--; //variableD becomes 4
```

# Calling Functions

Functions can be used to perform a group of instructions. They can be given data, as their *arguments*, and produce a value called a *return value*.

Much like you can place a variable in code to use that data it stores, you place a function in code to use the value it returns.

These must appear *inside* a function.

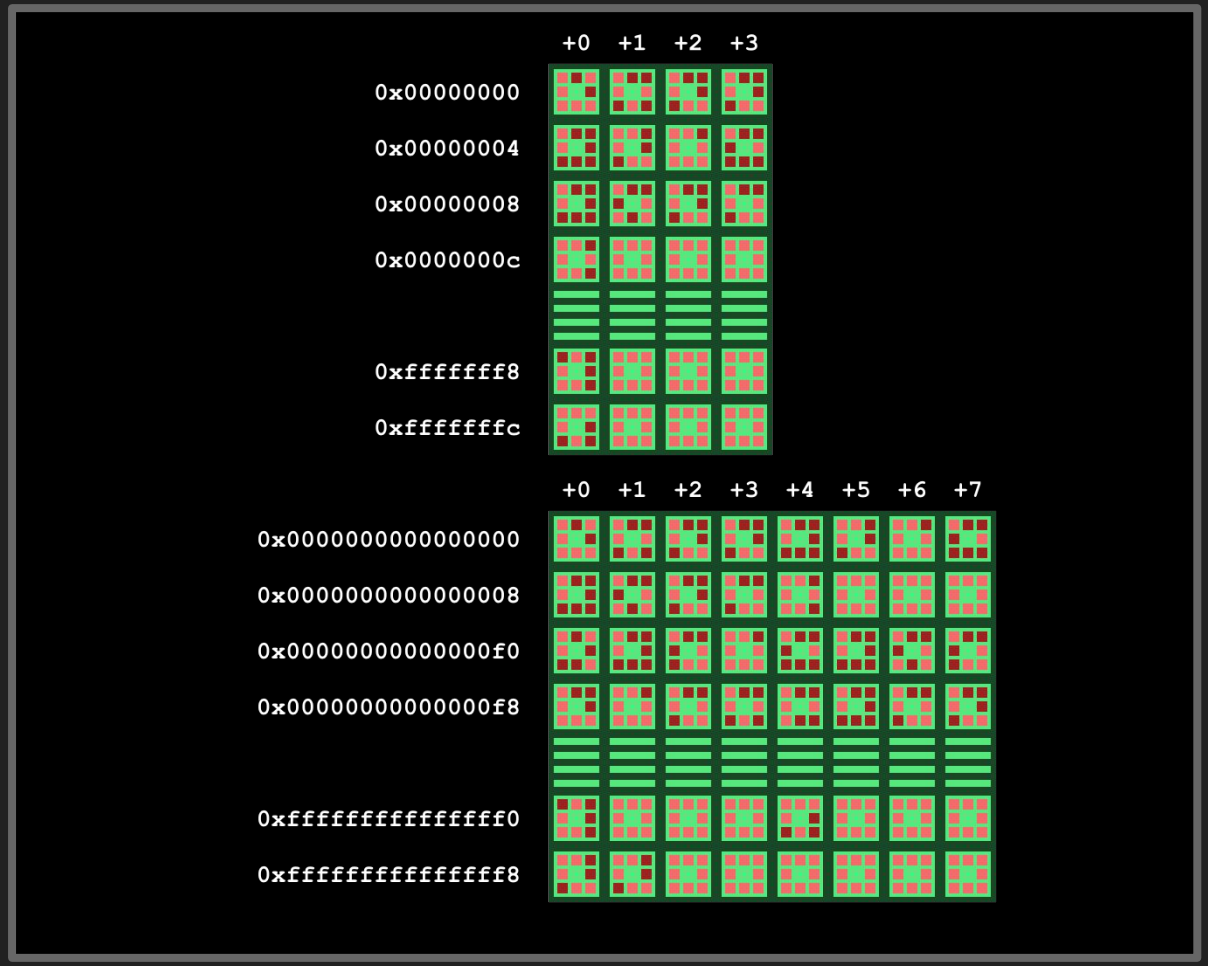
```
/*
 * This calls the function named `printf`. You need to provide
 * this function with its first argument as a string, describing
 * this format of what to output. Then all of the data to
 * output.
 */

yourNumber = 79;

// This will display `Your number was 79`
printf("Your number was %d\n", yourNumber);
// This will display `Your number was 4f in hexadecimal`
printf("Your number was %x in hexadecimal\n", yourNumber);
```

Now we have enough to make a simple program

# Memory

In modern processors, memory is treated as one long set of *bytes* (each containing 8 *bits* of data). Each *byte* as it's own *address* that we can use to reference it, which is just the number of bytes it is from the first byte in memory.

Memory is also split into *words*, which are the sizes of data that the processor works with. For *32-bit* computers, *words* are 4 *bytes* long, and for *64-bit* computers, *words* are 8 *bytes* long.



# Memory

Different types use different numbers of bytes to store the data related to their type. Typical *32-bit* sizes for different types are shown.

Data types are aligned to *words* as shown, as processors move data around as entire words. For example, `int`s will occur across a single word, not split across two. `short`s will appear on either half of a word, not in the middle.

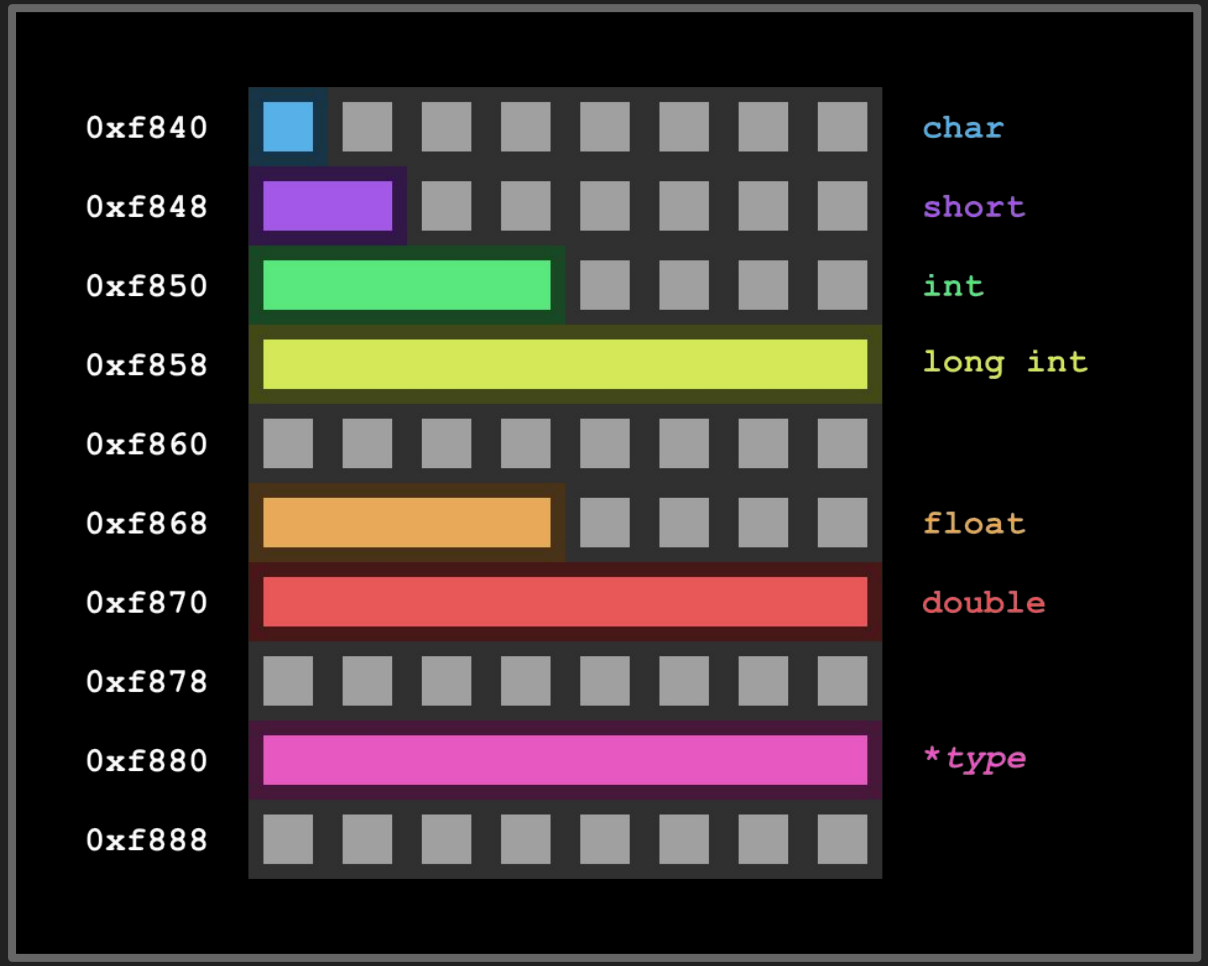
You can use `sizeof` to see how many bytes are used by a type or variable.

# Memory

Different types use different numbers of bytes to store the data related to their type. Typical *64-bit* sizes for different types are shown.

Data types are aligned to *words* as shown, as processors move data around as entire words. For example, `int`s will occur across a single word, not split across two. `short`s will appear on either half of a word, not in the middle.

You can use `sizeof` to see how many bytes are used by a type or variable.



# `printf` Format Strings

You can control how data gets displayed when using printf using the format strings shown.

They all work on numeric numeric int type only.

You can use `%d` for decimal (base 10), `%x` for hexadecimal (base 16), `%o` for octal (base 8), and `%c` to print as an ASCII character.

```c
/*
 * These are the following `format flags`:
 * They all work on the `int` numeric types
 */
int x = 65;

printf("This is a decimal number: %d\n", x);
// This is a decimal number: 65

// Assumes unsigned
printf("This is an octal number: %o\n", x);
// This is an octal number: 101

// Assumes unsigned
printf("This is a hexadecimal number: %x\n", x);
// This is a hexadecimal number: 41

printf("This is an ASCII character: %c\n", x);
// This is an ASCII character: A
```

# `printf` Format Strings

You can also add characters before `d`, `x`, or `o` to allow you to work with types of different sizes.

Use `ll` for `long long int`, `l` for `long int`, `h` for `short`, and `hh` for `char`.

```
/*
 * You can use the following prefixes to work
 * with types of differing sizes.
 */

long long int a = 0xAABBCCDDEE;
printf("long long int is %lld\n", a);
// long long int is 733295205870

long int b = 131071;
printf("long int is %lx\n", b);
// long int is 1ffff

short c = 41146;
printf("short is %ho\n", c);
// short is 120272

char d = 88;
printf("char is %hhd\n", d);
// char is 88
```

# Binary and Hexadecimal

It is helpful to be able to convert binary to hexadecimal as sometimes we want to see how a number or value is represented in binary, but having the computer write hexadecimal uses less space.

Hexadecimal is used as one hexadecimal digit represents exactly 4 binary digits or *bits*.

A set of 4 *bits* is often referred to as a *nibble*.

| Decimal | Binary | Hex | Decimal | Binary | Hex |
|---|---|---|---|---|---|
| 0 | 0000 | 0 | 8 | 1000 | 8 |
| 1 | 0001 | 1 | 9 | 1001 | 9 |
| 2 | 0010 | 2 | 10 | 1010 | A |
| 3 | 0011 | 3 | 11 | 1011 | B |
| 4 | 0100 | 4 | 12 | 1100 | C |
| 5 | 0101 | 5 | 13 | 1101 | D |
| 6 | 0110 | 6 | 14 | 1110 | E |
| 7 | 0111 | 7 | 15 | 1111 | F |

# `printf` Format Strings

You can print real numbers using `f` for `float` and `lf` for `double`.

You can also print exponential or scientific form using `e` for `float` and `le` for `double`.

```c
/*
 * You can print real or floating point
 * numbers.
 */

float x = 12.5;
double y = 13.25;

printf("float is %f\n", x);
// float is 12.5
printf("double in exponent form is %le", y);
// double in exponent form is 1.335000e+01
```

# `printf` Format Strings

There are a number of other ways you can control output using characters immediately after the `%`, such as `#` for alternate representations, `0` to use `0` as padding, and `+` to force the output to show its sign.

You can then add a number to force it to take up a certain number of character spaces. Use a `-` before the number to align it to the left of this space.

Also, use `%%` to just print a `%`.

```c
/*
 * You can modify how each is printed
 */

int x = 65535;
int y = 4095;
double z = 12.13;

printf("x is %d, y is %d\n", x, y);
// x is 65535, y is 4095

printf("x is %#x, y is %#o\n", x, y);
// x is 0xFFFF, y is 07777

printf("z is %12lf\n", z);
// z is    12.130000

printf("z is %+lf\n", z);
// z in +12.130000

printf("x is %08d\n", x);
// x is 00065535
```

# `printf` Escape Characters

Escape characters let us print characters that are hard to put into strings or that are hard to identify when typed.

Common escapes include `\n` for a newline (Windows uses the pair of chars `\r\n` for newlines), `\t` for tabs, `\x##` for a character with a specific hex value, and `\\`, `\"`, and `\'` for using `\`, `"`, and `'` in strings.

```c
/*
 * You can insert escape characters into any string
 */

printf("This line is printed\nabove this line.\n");
// This line is printed
// above this line.

printf("Windows uses\r\nto break lines.\n"
// Windows uses
// to break lines.

printf("This\tline\tuses\ttabs.\n");
// This    line    uses    tabs.

printf("This string ends\x00 prematurely.\n");
// This string ends

printf("You even need to escape \\.");
// You even need to escape \.

printf("You can even use \"quotes\" in strings.\n");
// You can even use "quotes" in strings
```

Let’s see what we can find out about different types

# User Input with `scanf`

`scanf` is very similar to `printf`, however it does the reverse. It takes a string which tells it what data to read in, disregarding whitespace and newline, and it stores that data in the given variable.

Instead of passing variables to `scanf`, you instead pass their address using the `&` prefix, which scanf uses to modify that data where that variable is stored.

```c
/*
 * You can modify how each is printed
 */

int x;
int y;
double z;

// 65535, 4095
scanf("%d, %d", &x, &y);

// 0xFFFF, 07777
scanf("%#x, %#o", &x, &y);

// 12.13
scanf("%lf", &z);
```

# Flow Control with `if...else`

The most basic way to control the flow of a program is with *conditional branching*. That is, perform one set of instructions if a condition is met. Sometimes an alternate set of instructions is called only if the condition is not met.

A value is considered to be *false* if it has the value of 0. All other values are *true*.

```
/*
 * You can branch based on conditions
 */

int x = 2;

if (x) {
    /*
     * The code in this block will run if
     * x is _equal to_ 2.
     */
} else {
    /*
     * This cose will run when x is
     * _not equal to_ 2.
     */
}
```

# Boolean Expressions

There are a number of operators used for *boolean expressions*. These are mostly for comparing two values.

Note that because `=` is used for assigning values to variables, `==` is used for comparing two values.

```
/*
 * Comparison operations have values
 */

int test1 = 2 >  0;     // 1 (true)
int test2 = 1 == 2;     // 0 (false)
int test3 = 3 <  3;     // 0 (false)
int test4 = 3 <= 3;     // 1 (true)
int test5 = 3 != 4;     // 1 (true)
int test6 = !(3 != 4); // 0 (false)
int test7 = !0;         // 1 (true)
int test8 = !1;         // 0 (false)
int test9 = !132;       // 0 (false)
int testA = !!132;      // 1 (true)

/*
 * You can combine the values of boolean expressions
 */
int andTest   = (1 <  2) && (3 >= 4); // 0 (false)
int orTest    = (1 <  2) || (3 >= 4); // 1 (true)
```

# Let’s respond to user input

# Type Conversion

Type conversion occurs when the data stored in a variable of one type either needs to moved into a variable of another type, either by assignment or as a parameter of a function, or when two items of different types are used in an operation.

The rules for type conversion may be confusing and difficult to understand, but it is important you understand how they work.

```
// Type conversion can happen automatically, such as with
assignment
unsigned short a = 65534;
signed short b = a; // b is now -2
int c = a; // c is not 65534

// Type conversion can be explicit (casting)
int a = (signed short) 65534; // a is now -2

// For operators, types are selected automatically, no smaller
than int
int a = (unsigned short) 65534 - (signed short) 4; // a is now
65530

// The above is automatically converted to, not because of the
type of `a`
// but because arithmetic requires int or larger types.
int a = (int) 65534 - (int) 4;
```

# Arithmetic Conversion Rules

For arithmetic, fractional types (`long double`, `double`, `float`) are preferred first. In the absence of these, everything is promoted to at least an `int`, with an `unsigned int` being considered wider than an `int`. Then, where an unsigned type is considered *wider* than a signed type of the same size, everything is promoted to the wider of the two options.

Note that if `long int` has the same size as an `unsigned int`, the `unsigned int` is considered the wider of the two, but an `unsigned long int` is used.

# Arithmetic Promotion & Conversion

Everything gets promoted to the highest on this list at least as far as unsigned int. Nothing should lose signedness here.

- `unsigned int`
- `int`
- `unsigned short`
- `short`
- `unsigned char`
- `signed char`

Everything is directly promoted to the widest of the types involved. Signedness is lost when both are the same size in memory and one is unsigned.

- `long double`
- `double`
- `float`
- `unsigned long long`
- `long long`
- `unsigned long int`
- `long int`
- `unsigned int`
- `int`

# Some Challenges

# Linux Manual Pages

You can find out a lot about different *standard library* functions from the `man` command with `man 3 function`.

This will tell you what header file it comes from, what its parameters are, what it returns, and related commands.

```
host:crashCourse$ man 3 printf
NAME
       printf,   fprintf,  sprintf

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int sprintf(char *str, const char *format, ...);

DESCRIPTION
       The functions in the printf() family produce output
       according to a format as described below. The function
       printf() writes output  to stdout, the standard output
       stream; fprintf() writes  output  to  the  given  output
       stream;  sprintf() writes to the character string str.
```

# Issues with `size_t`

There is a type that is commonly used by system functions that will be seen throughout this book, known as `size_t`. Importantly, wherever sizeof is used, it is evaluated as `size_t`.

For our systems, this is likely to be the same as a `long unsigned int`, which is standard for *x86_64* programs compiled with gcc or clang.

The conversion rules means that all integer literals become unsigned literals when used with sizeof.

```c
int main(void) {

    printf("Size of size_t is: %lu\n", sizeof(size_t));

    // Signedness check
    size_t test = -1;

    if (test < 0) {
        printf("size_t is signed\n");
    } else {
        printf("size_t is unsigned\n");
    }
}
```

```
Size of size_t is: 8
size_t is unsigned
```

```
sizeIssues.c:15:14: warning: comparison of unsigned expression
< 0 is always false [-Wtautological-compare]
    if (test < 0) {
        ~~~~ ^ ~
1 warning generated.
```