Deep C Dives
Adventures in C

Mike James

I/O Press
I Programmer Library

Mike James,
Deep C Dives

First Edition

ISBN Paperback: 9781871962888

ISBN Hardback: 9781871962215

First Printing, 2024
Revision 1

Published by IO Press	www.iopress.info
In association with I Programmer	www.i-programmer.info
and with I o T Programmer	www.iot-programmer.com

For updates, errata and links to resources,visit its dedicated page on the IO Press website: iopress.info.

Preface

C is a great language with a long past and probably a long future. The reason for this is that it occupies a unique niche in the ecosystem of languages. It is about as low as you can go without actually bumping into assembler. If you want to work with the machine that lies beneath all of the abstractions applied in other languages, C is the one to choose. The only problem is that most books and articles on the subject of C are written by programmers who would really rather be writing in one of those other languages. As a result you get a view of C that compares it unfavorably with modern, high-level, languages. There are exceptions, notably Harry Fairhead's two books on how to understand and use C on its own terms. Rather than a follow-on from Harry's work, this book can be regarded as a companion, an in-depth exploration of the essence of C and a statement of its distinctive traits.

To emphasize the way in which chapters of this book focus on specific topics, they are referred to as "dives", something that also implies a deep examination of the subject. Hopefully in each you will encounter some things that you already know but will come to see them differently. You will also read things that take you into the lesser-understood areas of the C language, which only make sense when you view C as a machine-independent assembly language. This is a good way to find out more about how the machine works and influences the way things are implemented. Other languages introduce abstractions to cover such inner workings and while this is usually a really good idea it isn't so much if your intent is to understand the machine or make use of the machine's hardware.

To view C in this way requires a complete shift in the way you view data. As languages have developed they have increasingly added more sophisticated data structures. These allow you to use data structures without worrying about how they might be implemented or how different ways of using them might influence efficiency. When you program in C any sophisticated data structures are down to you to implement and to do this you need to understand the basic data types that C provides.

For C the most important idea regarding data is that everything is a bit pattern and it is up to you how to interpret it. The options are wide ranging– a set of switch positions, positive integers, logical states, signed integers, text, hardware control lines, etc. This isn't a viewpoint endorsed by users of high-level languages who tend to see C as lacking the facilities to do a complete job.

What all this amounts to is that you can treat C as if it was a not quite up to the mark version of Java or Python or whatever high-level language you care to use, the result of which would be a deep disappointment in C. On the other hand, if you treat C as a language to solve the sort of problems it was designed to solve and stop comparing it to high-level languages then the result is not only efficiency but also elegance.

If you know some C, read on to find out what more you need to know to see it as its creators intended it to be. This reveals that it has a very special place among the programming languages of today as a powerful and versatile option for low-level programming.

Thanks to my tireless editors Sue Gee and Kay Ewbank. Programming is the art of great precision, but English doesn't come with a built-in linter. Errors that remain, and I hope they are few, are mine.

You can contact me at mike.james@i-programmer.info

Mike James
June, 2024

Table of Contents

The Prolog

C

“C is quirky, flawed, and an enormous success.”

Dennis Ritchie

There are lots of languages you could use, but C occupies a special place in the ecosystem of languages. Most attempt to “grow up” and occupy the higher plane of computer languages that abstract away from the hardware and move you toward the concepts of the problem you are trying to solve. C, on the other hand, stays with the hardware and lets you use it as you see fit to solve the problem. Of all of the available popular languages, only C provides this low-level support. This said, most of the higher-level languages, C#, C++ and Java in particular, borrow heavily from C in their basic syntax and this makes C seem familiar to an existing programmer. However, it isn’t just one of these high-level languages written differently and it isn’t a failed high-level language that one day will add the extra abstractions needed to allow it to emerge as something more modern.

C is already what it needs to be to fulfill its role as a modern, machine-independent assembly language.

Machine-Independent Assembler?

So C is not Java, is not C#, is not Go, and so on. While it is a crowded market among the high-level languages, if you want to work with the hardware free from constraints there really is only C. It is far from perfect, but it is realistically your only choice if you are looking for a language that works down at the level of assembler.

Not everyone thinks that this is so.

Even the statement that C is a machine-independent assembly language is much disputed and there are senses in which today this is less true than it was. This may be so, but there are more reasons that make it true than not. The argument against is that C no longer captures the way a modern machine works. This is true, but the loss of accuracy is minor compared to the abstractions applied to create other languages. C is still close enough to the machine for you to hear it creak.

The computers of a few years ago were simple compared to what today's hardware does. The basic computer of a few years ago would read instructions and obey them one after another until the program was complete – a single processor, in-order, execution.

Today's computers, on the other hand, use complex processing methods to try to speed things up. Pipelines process multiple instructions at the same time, caches keep the active part of the program close to the process and avoid having to go to slow main memory to read instructions and finally, yes, your instructions may not be obeyed in the order that you wrote them. A modern computer is a multi-core/threaded, pipelined, cache interfaced, out-of-order processor.

This is not your father's computer – but it doesn't matter.

The reason it doesn't matter is that all of these innovations are supposed to be invisible to the programmer. For example, cache memory speeds up reading and writing the main memory, but for most of the time you can forget it is present. Only when you start to consider optimization does how you access memory to avoid a cache miss come into the picture – and we all know the saying about optimization:

> *"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."* Donald Knuth

In the same way instruction reordering only becomes visible to a C programmer when the order affects the hardware in ways that are different. For example, when writing to two memory locations generally the order of the writes doesn't matter, but if the memory locations correspond to hardware registers that control some device then the order can be important. Again, most of the time you can ignore reordering. When it comes to using multiple cores – yes, you do need to think about this more often, but we still tend to reason in terms of a single thread of execution. We control how these interact using locks or some other methods which are also part of programming with C but not part of the C language.

What all this means is that while it is true that the execution model that C implies is not accurate for a modern computer it is good enough for most things. As a result C can be treated as a machine-independent assembler for a machine that actually uses other techniques to run the program faster.

C is machine-independent only in its core features. To cope with the specifics of a particular machine it generally needs either additional libraries or compiler extensions. This is more or less how it should be and attempts to absorb these into the main language have mainly failed. For example, ISO threading was introduced in C11 as a language standard – however, the non-language standard pthreads library is still used more.

Similarly, C11 introduced memory models that take care of instruction order, but most processors still require you to use custom libraries to control such things.

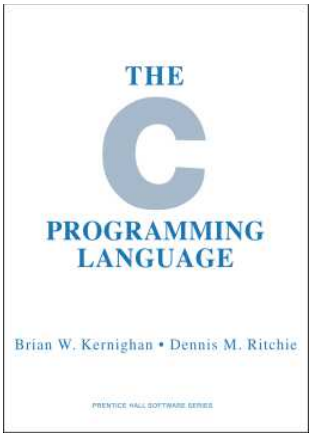
The core of C really uses a very simple, but appropriate, processor model.

The History

The C language was created in the early 1970s by Dennis Ritchie at Bell Labs. The language was invented to allow himself and Ken Thompson to create the Unix operating system for a PDP 11 mini computer. You can see that right from its initial creation the purpose of C has been systems programming and interacting with the hardware.

The C language didn't appear out of nowhere – there were a number of languages in the same niche that predated it. One of the most successful was BCPL, Basic Combined Programming Language, invented at the University of Cambridge in 1967. The BCPL language was notable for two things – the first being the use of {} to group blocks of code and the second, the way that the compiler itself was written in BCPL.

BCPL was converted into the B programming language by Dennis Ritchie and Ken Thompson. This was a typeless language that basically used the native word size of the machine it was running on either as data or an address. This evolved into C mostly by the addition of a range of data types to cope with the way ASCII text was packed into multibyte words.



It wasn't until 1978 that Kernighan and Ritchie published the first edition of *The C Programming Language*, a book which quickly became a classic and is usually just referred to as "K&R C". The book's simple presentation made the language seem even simpler and encouraged programmers to use it.

The spread of C compilers also made C a language that was easy to get hold of and one that promised easy porting between machines. The PCC Portable C Compiler became the *de facto* standard and was only replaced by the GNU C compiler in 1994. The compiler's machine-dependent parts were concentrated into a small number of source files, making it possible to create a C compiler for a new machine very quickly.

C also spread, though not as fast as Basic, on the early microcomputers of the 1980s. This was mostly due to the publication of "A Small C Compiler for the 8080s" in the then influential Dr. Dobb's Journal. This provided a listing of an entire C compiler written in Small C – it was a self-compiler.

As a result C became available on almost every machine and it was the language that was used to create systems programs that interfaced, or even modified, the operating system. All of the popular operating systems were, and still are, written in C – with the exception of Windows that mainly used C++.

Standards

Even back at the start of the 1980s it became clear that C was an important language and needed standardizing. Until then the language as described in K&R served as the standard, but by 1982 it had grown apart from the language as actually used in the real world. Compilers had added features that weren't in the original language – compiler "enhancements" are still a feature today and they tend to spoil the idea of a "standard" language.

Both the American National Standards Institute, ANSI, and the International Organization for Standardization, ISO, were involved in defining the C programming language standard. In 1989 the ANSI C committee, formally called the ANSI X3J11 committee, produced the first standard and the same standard was ratified by the ISO in 1990. Thus we have ANSI C or C89 and ISO C or C90 which refer to the same language.

The next important standard was C99, there is C95 in between, but this was mostly bug fixes to the previous one. Today we tend to think of C89/90 as "classic" C and C99 as modern C. Most programmers prefer C99 for two simple additions it introduced – the ability to use single line comments prefaced by `//` and the ability to declare variables in code especially in for loops. That is, before C99 you couldn't write `for(int i=0;` as the `int` had to be defined outside of code.

The next standard, C11, didn't do much for the language. It introduced an inadequate attempt at Unicode support together with standard threading that so far hasn't replaced pthreads and atomics. C17 was a bug fix that didn't introduce anything new. At the time of writing, May 2024, C17 is the current standard, although it is soon to be replaced by C23.

Currently most programmers prefer C99 as their standard as this is the best mix of classic features and modern extensions. The soon-to-be published C23 standard has some useful additions and looks to be the first refresh to have an impact on day-to-day C programming.

Compilers Today

Today the dominant C compiler is part of GCC, the GNU Compiler Collection. GCC was first released in 1987 by Richard Stallman when it consisted of a single C compiler. Since then it has been extended to support C++, Go, Rust, Fortran, Ada and others. It has been ported to a very wide range of machines and it is generally regarded as the standard C compiler.

What is surprising is that Windows lacks an official port of the GCC compilers. There is also no Microsoft C compiler as the compiler used to create Windows, Microsoft Visual C++ (MSVC), is first and foremost a C++ compiler. This can be used to compile C programs but it misses some details of the C11 standard and has only recently caught up with C99. You can either opt to live with MSVC or you can install one of the Windows ports of GCC.

The only real competitor to GCC is Clang, which was developed by Apple as it could not accept the GCC licensing terms. Clang takes an interesting approach to implementing a compiler and some users prefer it. On balance, it doesn't offer much over GCC in terms of compilation speed and the code it produces is often not as fast. The Linux Kernel is still compiled using GCC despite efforts to make it compatible with Clang.

In most cases the compiler that you want to use is GCC or a version of GCC modified for the hardware you are using.

It is worth saying that modern compilers, GCC in particular, are excellent at producing optimized code. I have often decided that some small piece of code needed to be written in assembler to gain maximum advantage of the machine only to discover that the code generated from a C program doing the same thing turned out to be faster. On examination of the generated assembler you generally find that the compiler knows all the tricks you do and usually a few that you don't and occasionally one or two that are hard to fathom. However, optimization sometimes goes too far. See Dive 16 on undefined behavior for more details.

C++

Of course, whenever you discuss C the elephant in the room is C++. This isn't the only object-oriented extension of C, Objective C is another. However, C++ evolved from C in a gradual fashion and its implementation has only slowly separated from C. The derivative language was first released by Bjarne Stroustrup in 1985. At first it really was an extension to C in the sense that it used a pre-processor to convert the text of a C++ program into pure C that was then compiled using the usual C compiler.

At the start C++ was no more than a text processor and hence highly compatible with C. Any C instruction was a valid C++ instruction and any C++ instruction was transformed into C.

As time went on, C++ developed more features and eventually gained its own compiler. Over time C++ and C diverged and today there are many things that are valid in C and invalid in C++ and vice versa. C is no longer a subset of C++.

The temptation when writing C is to think that it is only a small change to introduce some features of C++ to gain some of its advantages. The problem with this is that C++ has grown to such a point that there is generally more than one way of doing anything and some of these ways are confusing and potentially more dangerous than simple C. This is such a problem that to this day Linus Torvalds will not allow C++ into the Linux kernel.

To quote:

> *"C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do *nothing* but keep the C++ programmers out, that in itself would be a huge reason to use C."*

I don't completely agree with the "substandard programmers" part of the quote, but I think that C++ can turn a reasonably good programmer into something worse. The main problem with C++ is that it isn't as suited as C to low-level tasks and it isn't as good as countless other languages at high-level tasks. It is a confused mix of low- and high-level. Like Linus you would do well to concentrate on C if your requirements are low-level programming interacting with the hardware.

Being True to C

C is just one level removed from assembler. It is easier to use and, given a good compiler, just as fast. So, despite what some programmers may claim, C is still a good more or less machine-independent assembler. The programmers who claim it is not are really not seeing the soul of C. They see the language as lacking the abstractions that other languages have and they think that C would be better with them rather than without them. There is an almost irresistible pressure to move all languages towards some pinnacle of theoretical abstraction. It is why modern languages tend to be very similar. As Dennis Ritchie once remarked:

> *"When I read commentary about suggestions for where C should go, I often think back and give thanks that it wasn't developed under the advice of a worldwide crowd."*

C is about bit patterns and moving bits around. At its most abstract it is about integers. There is even an argument that if you are using floating-point types you probably shouldn't be using C. If you don't want to be close to the machine then choose another language.

The important point is that you will read many definitive proclamations about how C should be used or written – usually by proscribing exactly what you should not be doing. Most of the time these pronouncements are misguided or motivated by considerations that are simply not true to the idea of C. Many of them originate with C++ programmers who have a different view point and a different level of abstraction. They often think that as C is almost a subset of C++ their opinions are appropriate but they are often really aimed at the needs of the C++ programmer.

How to Read this Book

This isn't a primer or a tutorial, it is a collection of deep dives into important topics that make C a unique language. There is no particular path through the book that you are expected to follow – just dive into topics that you want to know about. Some dives refer to topics that are covered more deeply in other dives. While I have tried to achieve an orderly presentation of topics in C, some forward and backward references are unavoidable, but this shouldn't be problematic if you already know some C.

The purpose of the book is to reveal the unique aspects of C and in doing so help you use C as it was always intended to be used – as a low-level language that excels in dealing with data at the level of the bit pattern.

Dive 1

All You Need Are Bits

"Bits are the raw material out of which we fashion the world."

If there is a single characteristic that sets C programmers apart, it is an understanding of bits. If you don't understand bits then you are a programmer in some other language pretending to be a C programmer. Harsh, but true. Most accounts of bit patterns start off from binary values then move on to two's complement and finally, sometimes floating-point, but this simply hides the fact that bits aren't any of these things. Let's start at the beginning.

States

A bit is a two-state system – a bit can be a `1` or a `0`. It doesn't matter what you call these two states, true and false will do, but so will up and down or on and off and many more. Computers work in terms of groups of bits which can be accessed and worked with as a single unit. Typically the size of the group is eight bits, that is a byte. While this isn't strictly necessary, we'll use it in our examples which would be similar for a different number of bits.

The eight bits in a byte are usually have a particular order and are generally numbered starting from the right:

```
7 6 5 4 3 2 1 0
x x x x x x x x
```

Bit `0` is generally called the "lowest" or "least significant bit" and bit `7` is called the "highest" or "most significant bit". The reason for this terminology comes from regarding the group of bits as a binary number. Notice that we need to make an "endian" decision. Which end to number bits from is arbitrary and depends on the machine's architecture, but this big-endian numbering is almost universal. The reason for this will become apparent when we deal with place value systems later.

Any bit in a byte can represent one of two states. For example, you could represent the state of a set of eight lights one per bit position corresponding to on and off. Storing a bit pattern into the byte would set the lights to a particular state and reading the bit pattern would give the state of the lights – notice no numbers were involved in this transaction.

Consider now what happens if a room has two lights in it, a left and a right. Now we have four possible states:

```
left light off right light off      0 0
left light off right light on       0 1
left light on  right light off      1 0
left light on  right light on       1 1
```

So, if you have four states to represent, we need to use two bits, eight states need three bits, 16 states four bits and so on. If you have a number of states that doesn't fit exactly into n bits then you simply combine some of the extra states. For example:

```
no  lights on    0 0
one light  on    0 1 or 1 0
two lights on    1 1
```

is how you might represent three states using two bits, but there are alternatives.

Again no numbers are involved in this – only bits.

Specifying Bit Patterns

Some bit patterns are thrust upon us from the outside world and many of them are specified as literals. A literal is just a value specified at compile time that the compiler converts to a specified bit pattern. We use numeric literals or constants so often that we almost ignore them.

For example:

```
int a = 42;
```

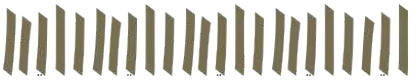
seems simple and innocent, but the compiler reads the `42`, the literal, and converts it into a bit pattern 00101010 and stores this in the variable `a`. There are many ways to specify a bit pattern using a constant – decimal, octal, hex and so on. For now all that matters is that you realize that in C every number you specify is converted into a representation that gives a bit pattern.

The next question is how numbers become bits and vice versa.

Place Value Systems

Bits are our raw material and we don't need to start from numbers to find them useful. In fact, numbers are just another "something" that bits can represent. However, unlike simple states, representing numbers requires a bit more organization. If you know everything about place value systems skip this section, although it does point out some things that more mathematical approaches miss.

Consider how you might keep a record of how many cars have passed your front door. You might well use a mark to indicate each one as it goes by to avoid getting confused:



but after a while the number of marks would become too many to cope with. In most cases you might start grouping the marks together – perhaps five at a time:

and you might use a special symbol for five cars:

```
V    V    V    V    111
```

where V represents a group of five ones. Congratulations, you have now invented the Roman system of number representation.

Your next obvious step as the number gets bigger is to invent another symbol for a bigger grouping. For example, X for ten items, which means we can now write:

```
XX111
```

to represent 23. We will ignore many ways the Romans invented ways to make their system simpler – for example 1X is a valid representation of 9 things as the small value to the left of the big value is subtracted from the big value.

This isn't a bad way to represent numbers until you try to do arithmetic and then you quickly discover its limitations. Try adding the number of cars I saw yesterday (27) to the number today:

```
(XXV11) + (XX111) = (XXV11XX111) = XXVXXV = XXXXVV = XXXXX
```

there is much regrouping of symbols to cope with.

If you think this is cumbersome try subtraction and then multiplication. For example, what is `XXV11` times `X`? It turns out to be easier to convert the Roman numerals to decimal notation, do the sum and then convert back.

Let's suppose for a moment that you are a Roman trying to make the system better. The key principle is that you use symbols to represent groups of items. So `I` is one item and `V` is a group of five. Clearly to represent a larger number you need to make bigger groups. Why not just repeat your first clever idea and use `V` to represent a group of five things, but why not have each thing be a group of five! This is the sort of thing a programmer would invent. So:

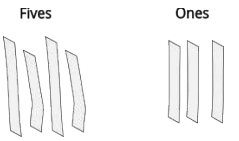
```
V = (IIIII)
```

and

```
V = (VVVVV)
```

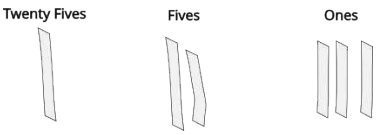
The problem is working out which of these you have when you encounter a `V`. The less creative thing to do is just invent another symbol and we are back to the Roman method. The really clever idea is to use the position that you write the symbol in a group of symbols. This is clever because, not only do you not need to invent another symbol, you can get rid of the symbol you just invented as well.

So you might record:



which is four lots of five and three lots of one i.e. 23 in our standard notation. Notice that you now don't need the new symbol `V`. You are counting in groups of five but you only need a symbol for numbers up to five because when you hit five you simply record a one in the next place.

This is such a good idea you might as well carry on with it. Why not add another position to count groups of five lots of five things, i.e. 25 things in our standard notation. For example:



So we now have one lot of 25, two of five and three ones making a total of 38 in our notation. Notice that again we still don't need the symbol for five and we don't need a new symbol for 25 – the place value system just works with the symbols we have. When you reach four lots of five and four lots of ones adding one more simply gives you one lot of 25 and a mark in the third position.

You may not need a symbol for five but you do need a new symbol. You need to be able to mark an empty position i.e. to mark that there are no groups of one, five or 25. You could just use a space but we adopted the standard zero 0 to indicate that there are no groups of the specified size. So:

is one lot of 25, no lots of five and three ones, i.e. 28.

This is how the Romans might have invented the place value system if only they hadn't kept on inventing symbols for ever bigger groups. The only cost is having to maintain the positions of the symbols and invent a zero. The payback is that arithmetic is much easier as you can work with each position in turn. For example, to add the number of cars seen yesterday (27) to those seen today (23):

```
  1          0       11 +
  0       1111      111
= 1       1111    11111
= 1      11111        0
= 11         0        0
```

or 50 in our notation. Notice that the addition was performed one "column" at a time and then counts that are too big are moved to the next column, the "carry".

This idea generalizes and if we work with groups of `N` and use the place value idea the first position counts from `0` to `N-1`, the second from `N` to `Nx(N-1)`, the next to `NxNx(N-1)` and so on. We never need to use the symbol for `N`, but we do need a symbol for zero.

Our standard system is base ten, for which we have ten digits, the symbols 0,1,2,3,4,5,6,7,8 and 9. In this base each place counts from `0` to `9` i.e. one less than 10 and there is no "digit" for 10. If you use all of the places then using just the first you can count from `0` to `9`, using the first and the second gets you to `99` and using the third as well gets you to `999` and so on.

A less familiar system, hexadecimal or base 16, works in exactly the same way. In this case there are 16 symbols 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F and again there is no symbol for 16. The first place counts from 0 to F i.e. 15, the second counts lots of 16 from 00 to F0 i.e. 0 to 240 and FF is 255.

Binary

The word “digit” means a finger or toe as well as a symbol for a number and our choice of base 10 is because, including thumbs, our hands have ten of them, but computers don’t have fingers so they are easier to build if we use bits.

The real reason that binary, base two, is so much better than base ten is simply that in binary, arithmetic is simple and so much easier to implement using logic. If you want to make a mechanical calculator using base ten you need cog wheels with ten teeth (`0` to `9`) whereas for binary the cog only has to have two (`0` and `1`)! This makes the rules you need to do arithmetic considerably simpler than for base 10.

What we have just discovered is that if we want to use just two symbols to represent numbers then the best idea is to reuse them via a place value system. So one bit, `0` or `1`, can count up to one. This is not much use, but if we use two places the second bit represents two items and so `10` represents two and is amusingly confused with ten in decimal allowing people to make lots of money selling T-shirts saying:

> *“there are 10 types of people in this world, those who understand binary and those who don't”*

Using three places the third bit represents four items, the fourth, eight items and in general the nth bit represents 2^n. Recall that 2^0 is one.

Going the other way the first bit gives you how many 1s there are, the second bit gives you how many groups of 2 there are, the third bit gives you how many groups of 4 there are and so on.

This makes it easy to find out what number is represented by a set of bits interpreted as a binary place value as a decimal place value - just add up the powers of two.

For example:

```
101010 = 1x32+0x16+1x8+0x4+1x2+0x1 = 42
```

It is interesting to note that standard C before C23 has no way of entering a binary literal, which for a format so important is a strange omission. Some compilers, GCC for example, use `0xb0101010` for binary and this has been added to C23.

Converting from binary to decimal is trivial, going the other way is slightly more complicated. You have to start with the biggest grouping and see if the number contains one of those. If so you have a leading `1` and if not you have a leading `0`. Next you have to remove that group from the total and then try the same trick with the next smallest grouping.

For a 16-bit value we have:

```
int16_t num = 42;
int bit = 0;
for (int i = 0; i <= 16; i++)
{
    bit = num / (1 << (16 - i));
    printf("%d", bit);
    num = num % (1 << (16 - i));
}
```

This divides the number by a decreasing power of two to find out if it needs a `1` or a `0` and then reduces the number to the remainder. Recall that a left shift is the same as multiplying by two.

Of course, we don't need to convert decimal numbers to binary in this explicit way as the command:

```
int16_t num = 42;
```

does it for us without us even giving it a thought. The compiler doesn't store decimal `42` in the variable; instead it converts the decimal value into binary and stores it in `num`.

So we already have the binary representation of `42` stored in `num` and working out what the bits would be is crazy as we already have the bits stored in `num`! All we have to do to see the binary representation is to print the bits of `num` out one by one and in the right order. This is easy using a right shift and a one-bit mask – see Dive 5, Bits and More Bits:

```
int16_t num = 42;
int bit=0;
for (int i = 0; i <= 16; i++)
{
    bit = (num  >> (16-i)) & 0x01;
    printf("%d", bit);
}
```

If you are happy with hex you don't even need to split out the bits one at a time:

```
int16_t num = 42;
printf("%X\n",num);
```

Which displays `2A`, the hex for `00101010`. The beauty of hex is that you can convert it to binary and vice versa one symbol at a time. For example, `2` is `0010` and `A` is `1010` which gives the complete binary value `0010 1010`. Going the other way you can convert any binary value to hex working four bits at a time. That is `0010 1010` is `2 A` – hardly any math needed.

That is, when you store `42` in a variable it is converted to binary without you doing anything at all. Variables do not store decimal values, but the bit patterns that represent the decimal values using binary.

Binary Arithmetic

Of course, the reason that binary is so natural in C is because most computers use hardware that is binary. The reason for this is partly that hardware is easier to build with just two states, 1/0 or On/Off, than with more but it is also true that binary arithmetic is easier to implement than any other because its tables have very few entries. We have to learn at least nine multiplication tables for decimal but for binary you only need your one-times table.

In a modern computer, memory stores bits and hence, if you want to store a number, binary is the obvious choice. More to the point, however, is that the Arithmetic Logic Unit (ALU) does binary arithmetic. You give it two binary numbers and it will automatically spit out their sum, if that's what you want.

Place value arithmetic works by doing the operations within each place and then adjusting the values next door, using a carry or a borrow if necessary. When humans do it the procedure can be messy with annotations of carry or borrow, but when computers do it in binary it's simple and regular. Decimal addition needs you to remember a table that gives the result of adding any digit to any digit. Admittedly many of the entries are obvious – adding zero and one hardly need to be remembered and addition is symmetrical. Even so, remembering this table is what makes addition difficult for beginners. Compare this to binary – you really only have to remember what 1+1 is!

When you add two bits together you get a result and a carry to the next position:

```
A + B = R  C
0   0   0  0
0   1   1  0
1   0   1  0
1   1   0  1
```

If you know your logic tables, and you should, you can see that R is A XOR B and C is A AND B. In binary, addition is logic.

Things aren't quite this simple because the carry has to be added to the position to the left of the current bit. We can perform an addition using logical operations and shift as follows:

```
sum = a ^ b;
carry = (a & b) <<1;
```

`sum` now contains the sum of `a` and `b` but ignoring any carry bits. The variable `carry` now has all the carry bits in the correct position to be added to sum to give the final answer.

We can repeat the addition to add the carry bits:

```
temp = sum ^ carry;
carry=(sum & carry)<<1;
sum = temp;
```

The problem is that this operation may generate a carry and this has to be added in again:

```
temp = sum ^ carry;
carry=(sum & carry)<<1;
sum = temp
```

This has to be repeated until the carry is zero:

```
    int sum = a ^ b;
    int carry = (a & b) << 1;
    while (carry > 0)
    {
        int temp = sum ^ carry;
        carry = (sum & carry) << 1;
        sum = temp;
    }
    printf("%d\n", sum);
```

If you know how hardware adders work, you might be puzzled as to where in the hardware there is the equivalent of a while loop? A hardware adder is just a collection of logic gates. In hardware the loop happens because the logic is given time for the carry signal to propagate and settle down to a final answer.

What all this means is that while binary is a natural way to specify, interpret and talk about bit patterns, it is not the only option and it is important to keep in mind that a binary number is no more real than a bit pattern that represents a set of lights.

Rollover

Real hardware is limited in how many bits it can store. C has a range of different variable types to reflect this fact and each holds a given number of bits, usually 8, 16, 32 or 64 bits. There are many operations that increase the number of bits needed to represent the result. Adding one to the largest value that can be represented in 8 bits needs 9 bits to represent it, a single shift left of the same value needs 9 bits and so on.

What happens when an operation causes the value to exceed the number of bits available is something defined by the hardware, but C11 defines unsigned integer overflow as rollover or wrap. This is achieved by simply ignoring the fact that additional bits are needed. If you have a 4-bit value, `1111`, and you add one to it then the result is five bits, `10000`, but with rollover this is truncated to four bits, `0000`. You can think of rollover as simply allowing the extra bits to “fall off the end”.

Notice that, as long as you are using a standard C compiler, you can be fairly secure that unsigned integers will roll over. The reason for this choice is that rollover is actually useful in practice and it corresponds to the theory of modular arithmetic. As an example of how useful rollover is consider an array with `256` elements. You can start storing data in the array with an index of `0` and increment the index after each data item is stored. When the array is nearly full the index is set to `255`. Storing something at element `255` causes the index to roll over to zero and start using the array from the beginning again. This is called a ring buffer and it is very useful if you have to deal with data coming in at a fast rate and you need to store the most recent values.

An even more important application of rollover is to implement negative numbers.

Negative Numbers

So far we only have positive binary numbers and, obviously, we also need negative numbers. We could do what we do with decimal numbers and invent a negative sign and this is sometimes done – so called "signed magnitude representation". If you do this then the computer's hardware is going to have to be modified so that it adds negative numbers correctly. There is a much better solution that doesn't need any modifications or additional hardware.

The negative of a number reduces that number to zero when added to it. For example:

$4+(-4)=0$

Indeed this is the mathematical definition of the negative of a positive value. If two things add up to zero then one is the negative of the other. But wait! In the last section we discovered that adding one to 255 represented using eight bits results in a rollover to zero. That is:

$1+255=0$

as long as rollover occurs. This means that 255 can be regarded as the negative of 1 and by the same reasoning 254 is the negative of 2 and so on. If we are prepared to use half of the available range to represent negative numbers we don't need any special hardware to do subtraction – we can just add the negative number and rely on rollover to get the right answer. This is called "two's-complement representation" for reasons that will become obvious.

For example, if we only have three bits and assign half of the range to negative numbers we have:

```
bits   unsigned     two's
                    complement
000    0             0
001    1             1
010    2             2
011    3             3
100    4            -4
101    5            -3
110    6            -2
111    7            -1
```

Notice that there is no difference in how the bits are manipulated – it is purely a matter of how you interpret the bits. If you needed an example of how C programming is all about how you interpret the bits, there is none better! Also notice that the most significant bit can be used as a sign indicator – a zero indicates a positive number and a one a negative number.

For example, add `011` to `100` and the answer is `111` and you don't need to know if it is unsigned or two's complement. If the numbers are regarded as signed we have `3 + 4 = 7`. If the numbers are regarded as two's complement then we have `3 + (-4) = -1`. It makes sense and is correct and only the interpretation changes.

Why is this representation of negative numbers called "two's-complement"?

In this system the definition of a negative of `x` is that `x+(-x) = 0`, but this only works because of rollover. In the case of the 3-bit example you can see that another representation of zero is `1000`, i.e. the rollover. So you can write `x+(-x) = 1000`, or eight in decimal, which means that `-x` is `8-x`, which if you check the table you will discover is correct. For example, `-4` is represented by `8-4 = 4`. In general, if you have an n-bit representation the two's-complement representation of -x is 2^n-x and hence the name two's-complement.

There is a relationship between two's-complement and the logical not, see dive 5, or the one's-complement. In words:

```
two's-complement = one's-complement + 1
```

or in terms of C operators:

```
-x = 2^n-x = ~x+1
```

as long as all operations are carried out using n-bits with rollover. This is useful as it can be used to compute the two's-complement of any value.

The reason that this works is because the largest value that can be represented by n bits is 2^n-1 so you cannot work out 2^n-x directly as 2^n is too large. So you have to perform the subtraction in two steps:

`2n-x = (2n-1 - x)+1`

but 2^n-1 is just n ones and you can subtract x from this by simply inverting x i.e. it changes all the zeros to ones and vice versa. This is a simple NOT of the value and so we have:

`2n-x = (2n-1 -x)+1 = ~x + 1`

It is clear that rollover of unsigned binary values is essential to making two's-complement arithmetic work, but we now have some new behavior. A two's-complement result can fall outside of the range that can be represented in the number of bits in use. This is usually called “signed overflow”.

For example a two's-complement 3-bit value can represent values from 3 to -4 and this is its valid range. Now consider 3+1, what value should this be mapped to in a 3-bit representation? You cannot represent 4 in a 3-bit two's-complement and the bit pattern that you get corresponds to -4 which gives 3+1 = -4 which is strange to say the least, with two positive values adding to give a negative value.

Back in the early days of C there was no standard for how negative numbers should be represented, but with C99 two's-complement was adopted as the standard. This makes it very clear how two's-complement overflow behaves and 3+1 is -4 even if this doesn't make much sense. The problem is that C99 and C11 define such overflow to be “undefined behavior”, which is a crazy decision – as it is clearly well-defined even if it is slightly strange. In the days when two's-complement wasn’t standard this might have been reasonable, now it is just an unnecessary danger, see Dive 16 Undefined Behavior.

Things are a little more complicated than this account suggests because of the use of automatic promotion in expressions which can increase the number of bits in use without the programmer being aware of it. This can cause results that you might not expect as rollover doesn’t happen where it should. More about this in Dive 4 Expressions. For now what matters is that the nature of the bit sequence and the arbitrary nature of its interpretation is clear.

Dive 2

These aren’t the types you’re looking for

“These aren't the droids you're looking for.”

Ben Obi-Wan Kenobi

Types, love them or hate them you cannot ignore them. Strongly typed languages currently rule the earth and anyone who suggests that strong static typing isn’t a good idea is swiftly stamped on. C is a strongly typed static language, but this does not necessarily mean what you think it does. C types aren’t the types you might want or expect. This said, the concept of type in C fits in perfectly with its low level intent.

The Origin of Type

The concept of types in programming is both deep and shallow.

It is shallow when you first meet it and gets increasingly elaborate as one abstraction is added on top of another. At its most advanced, types provide a theory of programming in the form of the Curry Howard correspondence which views programs and mathematical proofs as being one and the same. This leads us into areas of pure mathematics that are generally labeled as category theory and logic.

The good news is that type in C is much more at the shallow end of the range but over time there have been efforts to make it deeper and hence more respectable and more like every other programming language. There is an argument that C needs a simple interpretation of type to be effective as a language that works at a lower level than most. Rather than abstracting type away from the way that the machine represents data, it is better to keep the association between data type and representation.

The fundamental, but not the only, idea of type in C is simply how many bits are in use and how those bits are to be interpreted.

This is a fairly rare interpretation of type these days and is another thing that makes C different.

C Types

There are only four fundamental types in C – integer types, floating-point types, pointers and structs. The first two types are all about how you interpret the bits – they are the scalar types. The third is about how you reference other types and the fourth is how you organize other types.

You might want to add arrays to this list, but an array is such a simplistic organization of other types that it almost doesn't count – it is basically a use of the pointer type, more of which later.

Word Size

At the raw machine level there are bits – nothing but bits. These bits are generally organized into chunks called nibbles, bytes, words, double words and quad words. The nibble and the byte are generally 4 and 8 bits respectively. The size of a word varies according to the machine, but essentially a word is the default size of a group of bits that the machine works with, usually 16, 32 or 64 bits, but these aren't the only possibilities. Machines in the past used word sizes of 18, 36 and 48 bits – which are rare choices today, but serve to remind that word size is fairly arbitrary.

Notice that word size is dependent on hardware. The electronics, the memory, the registers and the buses that connect them determine the word size. Machines generally aren't restricted to working with a single word size, but there is usually one that fits their architecture better than any other and this is usually the fastest and most efficient.

These groups of bytes are important because they are the units that the machine uses to process data. You can usually write instructions that work with bytes, words or double words and so on. For example, you could add two bytes together or you could add two words together and so on. At the machine level there would be distinct add instructions to add two bytes and two words and so on and you would have to work out which to use in any given situation.

When writing in C the compiler does the job for you and it selects the instruction according to the type of the variables you are using. For example:

```
char a = 1, b = 2;
char c = a + b;
printf("%d\n", c);
```

In most systems `char` is an 8-bit data type and so the compiler creates the machine instructions to add two 8-bit bytes. If you had written:

```
int a = 1, b = 2;
int c = a + b;
printf("%d\n", c);
```

then the compiler would have selected the instructions needed to add two 32-bit words – assuming that `int` is a 32-bit type for the machine in question.

That is, the fundamental reason for types in C is to determine the type of operation appropriate for the data.

Type determines how operators act and this is always true, no matter how much we elaborate the idea of type.

You could also add that, in C, type also determines how many bits are involved in the operation but this is more obvious – well it is to a C programmer...

Data Representation

Data representation is a key emphasis of C as opposed to other languages which try to hide how they represent data. The trend in ever higher-level languages is to abstract away from the underlying bits and encourage the user to think in terms of objects that they can use without knowing how they work.

We have seen that in C it is the bit sequence and what it represents that is important. In other languages bits, representations and their limitations are hidden. For example, JavaScript just implements a number object and handles the distinction between integers and floats under the covers. Python has integers with unlimited precision so you never need ask what is the range of a Python integer. C on the other hand insists that you know how the data is represented. It has different sizes of integers and even has unsigned integers and you are expected to know how these work and how they relate to one another. More accurately, if you don't understand the data representation in use you will find it harder to write correct code.

It is important to remember that, while the machine and C may offer a range of operations, which operations make sense is entirely dependent on what the bits are currently taken to represent.

This leads on to the interesting question of why a byte in C is called "char" and not "byte"? The reason is that back in the early days a char was intended to hold enough bits to represent all of the text characters the machine needed to work with. As most machines worked with ASCII, a seven-bit code, most often char was defined as being eight-bit to allow for some additional characters. Notice that right from this starting point it is clear that C wants you to have an interpretation for the bits it stores. You can add two chars together, but in this case you are supposed to think of them as eight-bit integers. You can also set all of the bits to one and then you are better thinking of them as light states.

In all cases we take a bunch of bits and combine them with another bunch of bits using an operator and the question is does the resulting bunch of bits make any sense.

The Usual Types

At this point you might hope that C defines types that are a fixed number of bits – not so. The problem back in the day was that different machines used different architectures and so you couldn't be sure that there was support for an eight-bit byte. As a result the char type was defined to be the smallest bunch of bits a machine could work with, which today is usually eight bits.

The next size up is "short". You might think that this would be 16-bits and it usually is, but it might just be bigger. The only thing you can say about short is that it should be able to hold at least 16-bits.

After short comes "int", which is the type we tend to use by default. It is defined to be at least 16-bits, but it might be bigger.

After int C becomes a little repetitive. We have "long" which is at least 32-bits and "long long" which is at least 64-bits.

In terms of number of bits we have:

```
char <= short <= int <= long <= long long
```

and in the far distant future we might have "long long long" and so on, but for the moment the set stops at long long.

Nowadays, a typical system has:

- char=8 bits
- short=16 bits
- int=32 bits
- long = 32 bits
- long long = 64 bits

In addition to these types you can also add "signed" or "unsigned" to the type to indicate if you want to store negative numbers or not. Originally C did not define how signed types were to be represented, but modern C insists on two's-complement form – see Dive 1.

If you choose signed then roughly half of the integer's range is negative and half positive. If you choose unsigned then the entire range is positive. For example, `signed int`, or just `int` which has "signed" as its default, has a range of −32,767 to +32,767 and `unsigned int` has a range of 0 to 65,535. You can see that if you were using a variable to count things, using `unsigned int` would let you count a lot more things than `signed int`. This is another example of it all being a question of how you interpret the bits.

Floats

As well as the integers, there are also generally three types of floating-point types. Of course a floating-point type is capable of storing a fractional value as well as whole numbers. Again this is another example of interpreting the bits. A float is a single precision floating-point number and on most systems it is 32-bits. Notice that a `float` and an `int` are both just 32-bit values. The difference is that for an `int` the bits are interpreted as a simple binary number and for a float the bits are divided into two groups – one giving the exponent and the other the fractional part. As well as float there is also `double` which is almost always a 64-bit value. On some systems there is also `long double` which can be anything from 80 bits to 128 bits but this isn't a common type.

There isn't much to say about floating-point types in C because they work just like floating-point types in any language – mainly because nearly all of them use the IEEE 754 standard. This means that using floating-point types in C has all of the problems of using them in any language – loss of precision and not doing arithmetic the same way humans do.

For example:

```
if(0.2+0.1==0.3){
    printf("correct\n");
}else{
    printf("incorrect!\n");
};
```

displays `incorrect` meaning that 0.2 plus 0.1 isn't 0.3. This is a consequence of 0.3 not being exactly representable in binary.

Floating point is about numbers and not so much bit patterns – if you are using nothing but floating-point in C then you might be using the wrong language!

The Usual Literals

A common requirement is to store a given bit pattern in a variable of the appropriate type. This is achieved using a numeric literal or constant. The compiler reads the constant and converts it to a bit pattern, which is then stored in the variable. Many C programmers don't bother specifying the type of a constant, but it is often a good idea and sometimes it is essential.

There is a set of suffixes that can be used to indicate the constant's type:

- l or L for long
- ll or LL for long long
- u or U for unsigned.

There is also the f suffix for floating-point.

If there is no suffix the constant is taken to be an `int`.

These suffixes were essential before C99, but after this constants take their type from what they are assigned to and they can generally be ignored. The C99 rule is that the type of a constant is the first type that is large enough to represent it from the list `int`, `long`, `long long`.

You can also use hex and octal constants. Hex constants start with `0X` and octal constants start with `0` and hence are error prone if you accidentally add a leading `0` to a decimal value. C23 has also added, at long last, binary constants which start `0b`.

If you create a constant that has more bits than required then the remaining bits are simply ignored. For example:

```
char a = 0x010203040;
```

stores `0x40` in `a`. You will probably see a warning from the compiler and perhaps from a linter, but it works.

If the constant has a sign then it is assumed to be a signed value and it is converted to two's-complement form. This can be assigned to an unsigned variable and the bit pattern will be unchanged, but its value will be interpreted as positive. For example:

```
unsigned int a = -1;
```

assigns `0xFFFFFFFF` to `a` as this is the two's-complement representation of `-1`.

One literal that might confuse you is a character literal, for example:

```
unsigned int a = 'C'
```

is an int bit pattern corresponding to the ASCII code for the character – see later.

Precise Types

Not being able to exactly specify the number of bits a variable can store is mostly a minor irritation, but there are arguments that making the size and signedness of a variable explicit is a good idea. For this reason many programmers define their own precise bit-size types and in C99 this was standardized.

You can use the type `int`*n*`_t` and `uint`*n*`_t` for a signed and unsigned integer with *n* bits respectively. The value of *n* is usually one of 8, 16, 32 and 64, but notice that if a system doesn't support a bit size then it simply omits the definition and your program will not compile.

Notice that the `int`*n*`_t` and `uint`*n*`_t` types are nearly always mapped to the older type specifiers on a given system, for example:

```
typedef signed char int8_t;
```

So, using precise types our previous program would be:

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

int main(int argc, char **argv)
{
    int8_t a = 1, b = 2;
    int8_t c = a + b;
    printf("%d\n", c);
}
```

Notice the need for the additional header file.

To provide even more flexibility C99 introduces `int_least`*n*`_t` which has at least *n* bits and `int_fast`*n*`_t` which is the fastest number of bits with at least *n* bits. Notice that in both cases you may get storage allocated with more bits than you specify and this alters when overflow occurs.

Notice that:

- `char` = `int_least8_t`
- `short` = `int_least16_t`
- `int` = `int_least_16_t`
- `long` = `int_least_32_t`
- `long long` = `int_least_64_t`

There are also two additional types that are occasionally useful:

- `int_max_t` the largest integer type available
- `intptr_t` gives enough bits to hold a pointer

The only use of `intptr_t` is if you need to cast a pointer to an integer to find out the address. In most cases you would be better off casting to `void*` unless you want to perform arithmetic or logical operations, i.e. integer operations on the address.

Char Type

All languages have to provide some way of converting a bit pattern into something that can be printed or displayed and C is no exception. However, C is, as you might expect, a little more low-level than most. It sets aside the `char` type as the smallest collection of bits that can represent the characters that can be displayed on the machine's peripherals. This is conventionally just eight bits and in most cases we can regard a char as a byte. By today's standards this is hopelessly out-dated because of the now almost universal use of Unicode which needs 21 bits per character or code point. To store raw Unicode you would need a char to be 32-bits, but this never happens. There is a `wchar_t` type but this varies in size from 16-bits to 32-bits and makes no mention of what encoding UTF-16 or UTF-32 should be used. In practice it is better to avoid `wchar_t` if possible and use UTF-8 – see later.

The most common way to work with Unicode across different languages and applications is to use UTF-8 which uses a variable number of bytes to represent a code point. The good news is that ASCII code is identical to single byte UTF-8 for the usual characters. The whole subject of how to work with Unicode in C is a difficult one and for the moment it is simpler to focus on standard ASCII output.

You may have a value stored in a variable:

```
int16_t a = 42;
```

This stores

```
0000 0000 0010 1010
```

in the variable – which is 32+8+2, a total of 42. To print this in a human readable form you have to convert this bit pattern into a set of bytes or chars that represent the ASCII or UTF-8 characters `4` and `2` which is `0x34` and `0x32` respectively. You can see that to convert to ASCII digits all we have to do is to add `0x30` to the binary value of `0x04` and `0x02`. If you tackle this as a programming problem you would probably come up with something like:

```
    int16_t a=42;
    char first= a/10 | 0x30;
    char second= a % 10| 0x30;
```

where `char` could have equally well have been `int8_t` on this machine. Of course, if the value has more digits than two you would need to continue in the same way, but this is the general idea. Once you have the two digits as char you can print them:

```
printf("%c%c\n",first,second);
```

which displays `42`.

Of course, you can achieve the same result more generally with:

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

The point of this example isn't so much to explain how `printf` formats work their magic, more that `printf` and much of what happens in C is about converting one representation into another. In this case the variable `a` was interpreted as holding a binary value and this was transformed into two additional bit patterns which represent the ASCII codes of the first two digits in the decimal representation.

This sounds convoluted, but it corresponds to what is going on and is how you should think about it. Only beginners think that

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

just “prints the contents of `a`”. It is also clear that we need to think about format specifications giving us two pieces of information – what the bit pattern in the variable represents and how it should be converted to an array of char. For example, the `“%d”` says that the bit pattern in `a` should be taken to represent a binary integer and it needs to be converted to an array of `char` that correspond to the ASCII characters of its digits.

The point is, when thinking about what C is doing you cannot ignore the way a bit pattern is modified and reinterpreted. This sort of analysis of behavior is very typical of C and hardly ever happens in other languages.

Printf and Type

As another example of how interpretations of bit patterns differ, consider the way format specifiers in `printf` and similar functions work. A little known and used feature of the print specifier is the length sub-specifier which explicitly gives the size, i.e. the number of bits to be used:

```
hh     signed char
h      short int
l      long int
ll     long long int
```

So for example to get `printf` to treat `a` as an unsigned short int you would use:

```
unsigned short a = -1
printf("%hu\n",a);
```

which displays `65535` as no promoting occurs. The only problem is that these lengths depend on the machine in use, i.e. `short int` might be bigger than 16 bits. There are some macros that are supposed to supply the correct format for precise bit sizes, e.g. `int16_t`, but they tend not to work by simply supplying `d` or `u` as the format specifier, rather than `hd` or `hu`.

You might be thinking that of course it displays `65535` as `a` is an unsigned short but this isn’t what is happening. The type of the variable doesn’t affect what is displayed. For example:

```
printf("%hu\n", -1);
```

also displays `65535`. You can argue that the `-1` is by default taken to be a signed int, but even so the `hu` format takes the last two bytes and interprets them as an unsigned short.

The situation with `printf` and the expressions that it converts is more complicated than it looks due to automatic promotion – see Type Casting.

Unicode

For a final example of how types and bit patterns work, consider the difficult problem of C and Unicode. While C doesn't have any standard Unicode features, C11 added the `uchar.h` header to provide some basic functionality, but it isn't universally supported. A much simpler solution is to use UTF-8 encoding because this is almost universally adopted by web pages, editors and display software.

UTF-8 is a variable length code that uses up to four bytes to represent a character. How many bytes are used to code a character is indicated by the most significant bits of the first byte.

```
Byte 1
0xxxxxxx   one byte
110xxxxx   two bytes
1110xxxx   three bytes
11110xxx   four bytes
```

All subsequent bytes have their most significant two bits set to `10`. This means that you can always tell a follow-on byte from a first byte. The bits in the table shown as `x` carry the information about which character is represented. To get the character code you simply extract the bits and concatenate them to get a 7, 11, 16 or 21-bit character code.

For example, if you have the bytes:

`11100010 10001000 10010001` or `E2 88 91` in hex

The first byte has `111` as its start and so this is a three-byte encoding. The second and third bytes start `10` so these are indeed the follow-on bytes. Extracting the remaining bits gives the 11-bit value:

`00010 001000 010001`
which is `2211` in hex, `8721` in decimal, and this corresponds to the mathematical summation sign, i.e. a sigma, ∑.

The first 128 characters of UTF-8 are the same as ASCII, so if you use a value less than 128 stored in a single byte then you have backward compatible ASCII text.

You might think that using UTF-8 was difficult from C without a UTF-8 library, but you can go quite a way with just the "it's all a bit pattern" approach. It is highly likely that the editor that you are using to create code is UTF-8 compliant, so if you know how to type extended characters into your program you can use Unicode literals.

Under Windows entering Unicode depends on the application you are using. A good approach is to enter the character using any application that works and then copy and paste it into your program.

Under Linux you have to press `CTRL+SHIFT` and while holding these down type a `u`. This will display u̲, a `u` with an underline. You can then type the

character code in hex and press ENTER when finished. As you type the character code you will see the hex displayed and you only see the Unicode character after you press ENTER.

For example:

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

int main(int argc, char **argv)
{
 printf("Sum Sigma %s\n", "∑");
}
```

The sigma is typed in under Linux using CTRL+SHIFT+u 2211 ENTER and under Windows it is better to copy and paste it. If you run the program you will see:

```
Sum Sigma ∑
```

Under Windows you will need to change the code page of the terminal window by using;

```
chcp 65001
```

to set the UTF-8 code page.

What is important here is not so much that Unicode with UTF-8 sort of works, it is the idea that UTF-8 is just another sequence of bit patterns. A C string is a sequence of bytes where usually each byte is the code for an ASCII character, but in most cases the functions that work with strings don't care what the bytes represent – they simply manipulate the bytes.

For example:

```
    char myString[] = "∑";
    printf("%s\n", myString);
    printf("%d\n", strlen(myString));
    char myString2[100] = "Sum Sigma ";
    strcat(myString2, myString);
    printf("%s\n", myString2);
```

It looks as if we have set myString to a single character, but this ignores the fact that as a UTF-8 encoding it needs the three bytes 0xE2, 0x88 and 0x91. The string displays perfectly as long as it is sent to a destination that accepts UTF-8 encoded data. The Linux terminal used with VS Code does, but the Windows terminal has to be set to UTF-8 mode by selecting a code page. The C program still knows nothing about UTF-8; it just sends the contents of the three-character string to the terminal for display. The next instruction displays the length of the string and it is indeed 3 and not 1. Next we concatenate two strings. Again the strcat function doesn't care that myString is UTF-8 encoded. As far as it is concerned it contains three

characters and these are added to the destination string and the whole string displays as:

```
Sum Sigma ∑
```

All of this working simply depends on the destination accepting and understanding UTF-8 encoding. Notice that if you save the string to a file and then reload it, – it just works.

So what doesn't work?

The answer is obvious. The `strlen` function reports the wrong string length. There are three bytes, but only one Unicode character. Similarly any attempt to modify the string has to accept the fact that a character is three bytes. You can see that you would need to modify the `strlen` and other character-oriented functions to take account of the UTF-8 representation. This isn't difficult. Unfortunately, Unicode isn't quite this simple. The simple-minded approach is to assume that each Unicode codepoint is a character – this isn't the case. Multiple codepoints can be put together to make a single character and the same character can correspond to multiple codepoints. Unicode has made text processing very much more difficult because you can no longer assume a simple one-to-one correspondence between a bit pattern and a character. In fact, Unicode has made things so complex that it is difficult to write a program that handles all of the possibilities correctly.

Final Thoughts

Other languages work towards complete data abstraction away from the hardware and the underlying bits that represent everything. C is one of the few languages that still maintains the connection between data and the bits that represent it. If you are working at that level and want maximum efficiency this is great. If not then data abstraction has its advantages. What is clear is that introducing more data abstraction into C is a bad idea.

We are C programmers - we work with bits.

Dive 3

Type Casting

“We cast a shadow on something wherever we stand.”

E. M. Forster

Casting or type casting is another of those defining qualities of C. Other languages may use the term “cast” but they don’t quite mean what C means. The idea of a C cast stems directly from the way C views data as bit sequences. There are only two fundamental or scalar types in C, integer types and floating-point types. The only real difference between them is the number of bits each can work with. Casting comes about because sometimes we need to change the number of bits and this is more tricky than it sounds. The problem is we would like the resized bit sequence to have a meaning as close to the original as possible but of course, this all depends on what the original meaning is taken to be.

Casting

In object-oriented languages casting has been extended to mean changing the type of an object, which generally involves no modification to the object. Casting in C seems to not modify anything, but then reality and convenience enter the picture and so does change. It also has to be said that modern C applies casts by default to help you get things done, but it is often better to explicitly write out a cast to make sure your intentions are clear.

The cast operation is simply a type name between parentheses.

For example:

```
(int) a;
```

In this example, whatever the type of `a`, it is cast to an `int`.

For a more concrete example:

```
char a = 42;
int b =(int)a;
printf("%d\n",b);
```

which displays `42` and it would be surprising if it didn't. However obvious this might be there is a lot going on. When you store `42` in `a`, which is a byte variable, the bit pattern stored is:

```
0010 1010
```

when you cast it to `int` the bit pattern is

```
0000 0000 0000 0000 0000 0000 0010 1010
```

assuming that `int` is four bytes. This may seem a small change but it is a change. The original bit pattern is extended by padding it with zeros.

When a smaller bit pattern is extended in this way it is called a "promotion" although the term upcast is also used.

Of course, sometimes you want to go the other way and reduce the number of bits in a representation. In this case you are using a "demotion", but this terminology isn't much used and downcast is also encountered.

For example:

```
int a = 42;
char b=(char)a;
```

The bit pattern in `a` is:

```
0000 0000 0000 0000 0000 0000 0010 1010
```

and this is demoted to:

```
0010 1010
```

in `b`. Of course, all that happens is that the bits that cannot be stored in the smaller variable are ignored. This can result in loss of data, but exactly what this means depends on what the bits represented. For example, if the bits were being used to record the state of lights on or off then a demotion would potentially lose the state of 24 lights. This might be fine if you only wanted to work with the most important eight lights, say.

When it comes to numbers, demotion has a simple and direct mathematical expression. The result of demotion to n bits is to take the original value mod 2^n.

For example:

```
(char) b = a % 256
```

Another way of looking at this is that if you demote to n bits then the result is what you would have got if the calculation had rolled over at 2^n – this is sometimes useful.

Negative Problems

So far it looks as if casting is trivial because all it does is add some zeros or chop off some bits but things are made slightly more complicated by the use of two's-complement to represent negative values. This is so common that we have to consider what happens when we cast a signed value. So far every thing we have examined applies to unsigned variables or signed variables that have a positive value. Things only get interesting for signed variables that have negative values.

Consider for example:

```
signed char a = -1;
int b =(int)a;
```

The bit pattern for an eight bit -1 is

`1111 1111`

If this is promoted to 32 bits by adding zeros this becomes:

`0000 0000 0000 0000 0000 0000 1111 1111`

which is **255**, i.e. a positive value. This is not good, as promoting an integer should not change its meaning – after all the 32 -bit value can certainly represent two's-complement -1.

The solution is to use a “sign extend” principle when promoting signed values. What happens is, if the cast is between signed values the top most bit of the smaller value is extended to fill the new bits in the larger value.

So when you do:

```
signed char a=-1;
int b=(int)a;
```

the bit pattern in `a`:

is promoted by sign extension to:

`1111 1111 1111 1111 1111 1111 1111 1111`

which is -1 in 32-bit two's-complement.

The important point is that a cast to a bigger signed value extends the “sign bit”, i.e. the most significant bit, be it a zero or a one.

Static or Dynamic Casting

Notice that now we have crossed a line. The original intent of casting was altering how we interpreted the bits with no changes to the bits. If you cast from signed to unsigned or vice versa with the same number of bits then no bits are changed in the cast.

For example:

```
signed char myVar1=-1;
unsigned char myVar2= (unsigned char)myVar1;
printf("%hu\n",myVar2);
```

This stores `11111111` in `myVar1` and this is interpreted as `-1` in a two's-complement representation. Assigning this with a cast to unsigned char stores the same unmodified bit pattern in `myVar2` and this is interpreted as `255`, a simple binary number. This example frequently crops up, but it demonstrates that a cast just changes the way you view the bit pattern.

It's only when we move to signed representations with different numbers of bits that we break the principle and modify the bits.

For example:

```
signed char myVar1=-1;
unsigned int myVar2= (unsigned int)myVar1;
printf("%u\n",myVar2);
```

Now the bit pattern:

`11111111`
is expanded to:

`1111 1111 1111 1111 1111 1111 1111 1111`

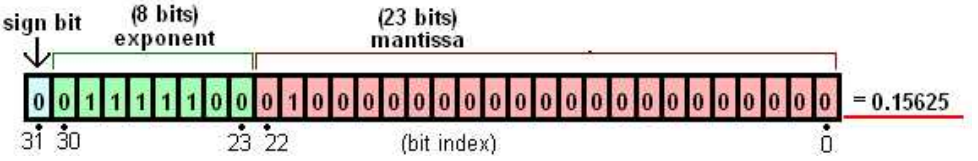
because of sign extension.

You could argue that, even for positive values, adding leading zeros is modifying the bits and you would not be wrong, but we now have the idea of manipulating the bits as part of a cast and this is strictly a process of dynamic type conversion, where the bit pattern is secondary to preserving its current meaning - which is arguably better handled by conversion functions.

The best example of this is floating-point casts.

Floating Point

The bit patterns used for integer types and floating-point types are quite different and the idea of casting one to another is slightly crazy if a cast means reinterpreting the same bit pattern. For example, a float uses four bytes to represent a sign bit, exponent and mantissa:



Clearly interpreting this as a 4-byte `int` isn't going to be a generally useful thing to do. If you try casting a `float` to an `int` you will discover what really happens:

```
float myVar1=41.9;
int myVar2=(int)myVar1;
printf("%d\n",myVar2);
```

The result is that `41` displays – the float has been converted to an integer and truncated, giving `41`.

If you try this with `-41.9` you will discover that you get `-41`.

This works the other way as well:

```
myVar1=(float)42;
printf("%f\n",myVar1);
```

and `myVar` contains `42.0`. Notice that it is possible to try to cast a `float` or a `double` to an `int` that cannot represent the value because it is too large. This is "undefined behavior", but in practice the most common option is to assign the largest integer value.

Finally, as a result of dynamic casting, there is no elementary way to gain access to the bytes that make up a float using casts. To find out how to do it we need to look at type punning and unions – see Dive 15.

Final Word

The idea of a cast starts out as being a way to change the interpretation of a bit pattern, but it quickly has to give in to the temptation of actively converting the bit pattern so that it represents the same quantity in the new type. Other languages give in and create a set of type conversion functions to do the job. C does have what look like type conversion functions, but they are actually arithmetic functions. For example, `round` converts a `float` to a `float` while rounding it to an integer. If you also want it to convert to `int` then you need a cast – if only an implied cast.

Dive 4

Expressions

"I don't know what the language of the year 2000 will look like, but I know it will be called Fortran."

CA Hoare, 1982

One of the most important concepts in any programming language is the expression and C has expressions for most things and knowing how to use them is a big part of effective C programming. What might surprise you is that there was a time when expressions didn't exist – they had to be invented. The generally accepted first implementation of expressions was in Fortran in 1957. John Backus and his team at IBM worked out how to compile expressions into machine code and the same algorithms are at work today, compiling C expressions into machine code. C A Hoare might have been wrong about what we call the language, but there is a little bit of Fortran in every modern language.

Expressions and Statements

There are only two types of instruction in C – expressions and statements. The simple difference between them is that an expression evaluates to a result and a statement doesn't. The distinction between the two is more subtle than this suggests, but if a result is returned then what you have is capable of being included in an expression, even if we don't usually think of the construct as an expression.

For example, a function call isn't usually called an expression, but it is if it returns a result. Statements are things like for loops, if statements and so on, essentially the control structures of C. A statement doesn't evaluate to anything in any situation and so isn't ever an expression.

So what is an expression?

The idea is an old one, dating back to the slow invention of how to write down arithmetic. Simple arithmetic has just four operators +, -, x and ÷, corresponding to addition, subtraction, multiplication and division. There are a number of possible ways to write down an arithmetic procedure, but the most common is to use infix notation where the operator is placed between the values it is to operate on.

For example, 2+3 evaluates to 5 and the plus operator + combines the 2 and the 3 by addition.

So far so good, but what about an expression like:

```
2+3×4
```

What does it mean?

We could interpret it as 2+3 gives 5 and then 5×4 is 20. This left-to-right implementation of the expression is perfectly valid, even if it is judged wrong by today's mathematical conventions in which multiplication has a higher priority than addition.

The correct implementation is to do the multiplication first i.e. 3×4 gives 12 and 2+12 gives the right result, 14 not 20. In the unlikely event that you are unsure about this you need to check up your arithmetic skills, not programming, as this comes from elementary math not the C language.

Notice that traditional mathematical notation is sophisticated in that it demands operations to be performed not in the order they are written.

This is what made it such a huge task initially to implement expressions in Fortran and what makes it difficult for beginners to learn how to work out even slightly complex arithmetic.

To make it easier for humans we invent mnemonics such as BODMAS – Brackets, Orders, Division/Multiplication, Addition/Subtraction, to give the priorities of the operations. Notice that brackets are always the highest priority – you have to evaluate all brackets before doing anything else. What this means is that even if you don't know the priorities of operators you can always force the order using brackets.

For example,

```
2+(3×4)
```

is the default order

and

```
(2+3)×4
```

is the left-to-right alternative order.

You don't have to know why we write arithmetic in this way to make use of it but once you have noticed the way it works you can't help but wonder why?

The answer is the distributive law.

Consider an expression like:

```
3×2+3×4 = 6 + 12 = 18
```

this gives the same result as:

```
3×(2+4)  = 3×6 = 18
```

We have noticed that we are multiplying both the 2 and the 4 by 3 and we have opted to do the addition first and save ourselves a multiplication by "pulling the multiply out of the bracket".

That a×(b+c) is the same as a×b+a×c is called the distributive law of multiplication and it is the reason we prefer to think of multiplication as having a higher priority than addition. If this wasn't the case and we evaluated 3×2+3×4 strictly left-to-right:

```
3×2+3×4 = 6 + 3 × 4 = 9 × 4 = 36
```

then the distributive law wouldn't be true.

In other words, if you want to think of:

```
a×b + a×c =  a×(b+c)
```

as "pulling a out into a bracket" then you need to make multiplication a higher priority operation than addition. This idea generalizes to the way other operators behave and their assigned priorities.

C Expressions

C has lots of operators that go well beyond the commonly encountered, standard ones. Once you start to extend the range of operators that a language has then you have to worry about associativity as well as precedence.

Associativity is simply the order that operators of the same priority will be evaluated. If you restrict your attention to simple arithmetic operators then the associativity isn't a problem because `(A+B)+C` is the same as `A+(B+C)`. However, when you move on to consider more general operators it does matter. For example, in C the right shift operator `A>>B` shifts `A` right by `B` bits. `(A>>B)>>C`, which means shift `A` right by `B+C` isn't the same as `A>>(B>>C)`, which shifts `A` right by the result of the right shift of `B` by `C` bits.

To make the result of such operations unambiguous, `C` defines each operator as associating left to right or right to left, i.e. right or left associative. The `>>` operator associates left to right because this means that `A>>B>>C` is `(A>>B)>>C` and this is the same as `A>>(B+C)`.

Every operator in C has a priority and an associativity and you need to know both to be able to figure out what an expression evaluates to. You can see all of the standard C operators in the table on the next page.

Precedence	Operator	Description	Associativity
1	`++`	Postfix increment a++	Left-to-right
	`--`	Postfix decrement a--	
	`()`	Function call a()	
	`[]`	Array subscripting a[b]	
	`.`	Element selection by reference a.b	
	`->`	Element selection through pointer a->b	
2	`++`	Prefix increment ++a	Right-to-left
	`--`	Prefix decrement --a	
	`+`	Unary plus +a	
	`-`	Unary minus -a	
	`!`	Logical NOT !a	
	`~`	Bitwise NOT ~a	
	`(type)`	Type cast (int) a	
	`*`	Indirection (dereference) *a	
	`&`	Address-of &a	
	`sizeof`	Size-of sizeof a	
3	`*`	Multiplication a*b	Left-to-right
	`/`	Division a/b	
	`%`	Modulo (remainder) a%b	
4	`+`	Addition a+b	Left-to-right
	`-`	Subtraction a-b	
5	`<<`	Bitwise left shift a<<b	Left-to-right
	`>>`	Bitwise right shift a>>b	
6	`<`	Less than a<b	Left-to-right
	`<=`	Less than or equal to a<=b	
	`>`	Greater than a>b	
	`>=`	Greater than or equal to a>=b	
7	`==`	Equal to a==b	Left-to-right
	`!=`	Not equal to a!=b	
8	`&`	Bitwise AND a&b	Left-to-right
9	`^`	Bitwise XOR (exclusive or) a^b	Left-to-right
10	`|`	Bitwise OR (inclusive or) a|b	Left-to-right
11	`&&`	Logical AND a&&b	Left-to-right
12	`||`	Logical OR a||b	Left-to-right
13	`?:`	Ternary conditional (see?: later)	Right-to-left
14	`=`	Direct assignment a=b	Right-to-left
	`+=`	Assignment by sum a+=b	
	`-=`	Assignment by difference a-=b	
	`*=`	Assignment by product a*=b	
	`/=`	Assignment by quotient a/=b	
	`%=`	Assignment by remainder a%=b	
	`<<=`	Assignment by bitwise left shift a<<=b	
	`>>=`	Assignment by bitwise right shift a>>=b	
	`&=`	Assignment by bitwise AND a&=b	
	`^=`	Assignment by bitwise XOR a^=b	
	`|=`	Assignment by bitwise OR	
15	`,`	Comma	Left-to-right

Compile v Runtime Expressions

The importance of expressions is that anywhere you can use a value in C you can use an expression to obtain that value. This is why you can write:

```
int a = 2*2;
```

or

```
int a = 2*b;
```

However, there is an important difference between the two. The first can be evaluated at compile time and the second cannot as we might well not know the value of `b`. This is an important distinction as sometimes a value needs to be known at compile time. For example, if you declare an array at file level then its size has to be known at compile time because it is allocated at compile time. What this means is that a declaration like;

```
int myarray[2*b];
```

will not compile unless it is within a function or a block. Notice that this is true even if the value of `b` can be deduced from the program, e.g. `b = 4`. A constant expression can't contain variables, increment or decrement operators or function calls.

All other expressions are evaluated at run time and their value isn't known until then.

Lvalues and Rvalues

Expressions produce results and results have to be stored somewhere. There are two sorts of objects involved in an assignment. The object on the left-hand side has to be something associated with an address where a result can be stored – this is generally called an lvalue. The object on the right of the assignment has to be a result, a value, i.e. something that can be stored in an rvalue. This is generally called an rvalue and so an assignment takes the form:

```
lvalue = rvalue
```

This is where the concept started, but over time it has evolved a little. Now an rvalue is something that has no obvious location in memory. For example, `a+b` is an expression the result of which isn't stored anywhere in particular. Clearly an rvalue can occur on the right-hand side of an assignment. In this extended sense an lvalue is anything that does have a clear identifiable location. You can also draw the distinction between modifiable lvalues and constant lvalues. In this sense an lvalue can be on the right-hand side of an assignment, but what is on the left has to be a modifiable lvalue.

The terms lvalue and rvalue are used generally, not just in assignment statements, and they often figure in error statements. For example, if you try:

```
&a = b;
```

you will often see the error reported as "`&a` is an rvalue".

Side Effects and Sequence Points

An expression should just return a result and change nothing else in your program. If an expression causes something else to happen then it is said to have a "side effect". In other languages the only concern is whether or not a function has a side effect. For example, a function that prints something has a wanted side effect, printing something. Functions that produce no side effects are said to be "pure" and functions in C can be pure, but often they are not.

What is surprising is that in C operators can have side effects and hence expressions can have side effects, even if they don't contain a function. For example, the unitary increment operator ++ when used as a postfix operator in `A++` returns current value of `A`. That is, it is the same as just writing `A`, but as a side effect it increments `A`. In other words, if `A` is `3` then after `B = A++`, `B` is set to `3` and `A` is set to `4`. Notice that this is quite different from `++A`, the corresponding unary prefix operator, the result of which is `A+1` and which has the side effect of incrementing `A`. So, if `A` is `3` then, after `B = ++A`, `B` is set to `4` and so is `A`.

Notice that the ++ postfix operator has the highest priority and associates from left to right, whereas as a prefix operator has a priority of 2 and associates from right to left. Also notice that `A++` has the same meaning as `A+= 1`.

It is this amazing richness of operators that makes C so attractive to anyone who has taken the time to master it. After a while you can write an expression such as:

```
C + = A++ + B++ + (D=++E) == F
```

What does this mean? Good question, so don't write expressions like this one.

Also notice that there are some problems with side effects. What does:

```
i = i++;
```

mean?

It all depends on when the increment is performed relative to the assignment. The problem is that the value of `i` is changed twice by the side effects of the expression, once by assignment and once by the postfix increment, and the order the side effects occur in isn't clear. To try to solve this problem C11 introduced the idea of a "sequence point", a point in the expression where all side effects so far encountered are forced to be complete. This provides an order for the side effects relative to the operators.

The end of any full expression obviously has to be a sequence point. The end of an initializer is also a sequence point and so is the comma operator and any function call in an expression. Notice, however, that the order in which the functions are called is not always specified and the order in which parameters are evaluated is never specified.

There is a sequence point after the evaluation of each format specifier in a `printf` to avoid any expressions you are evaluating interacting in undefined ways.

Logical OR and AND operators, `||`and `&&` are sequence points because they are evaluated in a different way to other operators. For an OR, if the left-hand expression is true then the right-hand expression is not evaluated because the result is already known to be true. Similarly for an AND, if the left-hand expression is false then the right-hand expression is not evaluated because the result is already known to be false. This is called "short circuited", or "lazy" evaluation. In both cases, however, the left-hand expression is complete, including side effects, before the right-hand expression is evaluated. Notice that if the right-hand expression has any side effects, including throwing an error, these will not happen if the evaluation is short circuited.

For example, in:

```
int result = a||b++;
```

`b` will not be incremented if `a` is true and in:

```
int result = a|| b/0;
```

no divide by zero error will occur if `a` is true.

For similar reasons, a conditional expression is a sequence point. The first expression is fully evaluated, including side effects, before the second or third expression is evaluated. That is, the `?` in `a?b:c` is a sequence point.

So far sequence points simply tell you when you can expect side effects to be complete, but they don't help with the problem of:

```
i=i++;
```

The sequence point is at the end of the expression and it still isn't clear whether the side effects occurred as assignment then increment, or increment then assignment. To deal with this the C11 standard also introduced two conditions that have to be satisfied to make an expression legal:

> *Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.*

These two rules make `i = i++` undefined as the variable has its value changed more than once.

The second condition also makes expressions such as:

```
array[i]=i++;
```

undefined as, even though `i` is only modified once, its original value is used to determine where the result is stored and not just what the result is.

Many programmers find the C sequence rules difficult to understand and apply, but this really isn't the problem. If your code causes you to contemplate the role of a sequence point or the conditions that apply then you are probably writing expressions that should not be written.

Auto Promotion

In an ideal world you would choose the types in an expression and expect the expression to be evaluated using them in the most efficient way possible. However, hardware isn't sufficiently flexible to make this a realistic possibility. In all cases hardware has a favorite type to work with. The key factor is the width of registers in the ALU, Arithmetic Logic Unit. Even 64-bit computers tend to process 32-bit data more easily than 64-bit values. Being 64-bit is more about supporting 64-bit addressing than 64-bit data.

In general, int is the type that should be the best for arithmetic operations on any given machine. On most modern machines it is 32 bits, but it's not that long ago that 16 bits was the norm.

What happens when you want to work with two data types that are smaller than `int`? To make things more efficient, they are auto-promoted to int following the usual casting rule – i.e. unsigned types are zero-extended and signed types are sign-extended. For example:

```
unsigned char myChar = 255;
int myInt;
myInt = myChar*255;
printf("%d",myInt);
```

In this case the result `65025` is printed because, although the arithmetic should overflow, both the values are converted to `int` before the calculation is performed. The expression is evaluated as if it was:

```
myInt = (int) myChar* (int) 255;
```

If there are types "larger" than int then casts are implicitly performed to promote everything to the largest size involved in the expression. The order is:

```
int < long<  long long < float < double < long double
```

and the unsigned types have the same rank as the corresponding signed type.

The rules, known as “the usual arithmetic conversions” seem complicated. The C99 standard says:

1. *If both operands have the same type, then no further conversion is needed.*
2. *Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.*
3. *Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.*
4. *Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.*
5. *Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.*

The most worrying part of this specification is point 3 as converting a signed integer to an unsigned of the same rank seems to invalidate the arithmetic. You have to remember that two’s-complement arithmetic works even if you consider the value to be unsigned.

For example:

```
unsigned int myUnsigned = 1;
int myInt = -1;
myInt = myInt+myUnsigned;
printf("%d \n",myInt);
```

In this case the `signed int` is converted to `unsigned` and it looks as though we are going to get the answer `2` as the negative is lost. This is not how it works. The `signed int` is converted to an `unsigned int` with the same bit pattern. When added together, the rollover occurs and the result is the same in two's-complement. That is, it prints `0` and we have the correct answer.

This doesn’t mean that you cannot get the wrong answer. For example:

```
unsigned int myUnsigned=UINT_MAX;
int myInt=1;
myInt=myInt+myUnsigned;
printf("%d \n",myInt);
```

In this case you will still see zero printed, which in this case is probably the wrong answer. Again we have the same bit patterns. The `UINT_MAX` is `-1` when regarded as a `signed int` and hence the final result is zero.

To get the correct answer we need to explicitly use a cast to `long long`:

```
unsigned int myUnsigned = UINT_MAX;
int myInt = 1;
long long mylong = (long long)myInt+(long long)myUnsigned;
printf("%lld \n",mylong);
```

In fact, we don't need to cast `myInt` to `long long` because the rules will promote it to the same type as `myUnsigned`.

Notice that these casts are only performed in expressions, including comparison operators, and bitwise operators. They are not performed for assignment operators. In most cases the result of expressions involving mixed types give you the correct answer but if in doubt use explicit casts.

A common idiom, however, is to rely on the rules to change integer arithmetic into floating-point arithmetic. For example:

```
int myInt=1;
float ans=myInt/100;
```

gives the answer `0` as integer division is performed before the result is converted to a `float`. However:

```
int myInt=1;
float ans=myInt/100.0;
```

gives the result `0.01` as the literal is now a `float` and hence `myInt` is automatically converted to a `float`.

Some Oddities

The whole idea of an expression is so obvious that you almost certainly don't need a conducted tour of all of C's operators and how they interact. However, there are a few strange and almost unlikely things that you might not have noticed. None of them have the advantage of clarity of purpose, but you will encounter them in other people's code.

Assignment is an operator

When you write `a=b` this is an expression just like `a+b` and the result is the value stored in `b` or the value of the expression on the right hand side. That is, `a=b` takes `a` and `b` and produces the result `b`. As a side effect it also sets a to the value stored in `b`, which is what you usually assume assignment to do! Notice that this implies that `a` cannot be an expression in its own right.

What advantage would this have?

The most obvious is that you can use it to set more than one variable:

```
a = b = c = 2*2;
```

which sets `a`, `b` and `c` to `4`.

You can use variables, even ones that are being assigned, in the expression:

```
int b = 3;
a = b = c = 2*b;
```

as the expression is worked out before the side effect is enacted. This sets all of the variables to `6`.

Another use is to save a result while computing a conditional. For example:

```
    int a,b = 0;
    while( (a = 2*b) < 10){
        b = b+1;
    }
```

The assignment within the while loop sets `a` to `2*b` and evaluates to `2*b`, which is then compared to `10`. At the end of the loop you will find that `a` is `10` and `b` is `5`. It is arguable that this is confusing, but it does avoid having to compute the expression twice. In short, you can test an expression and save its value.

Notice that all of the assignment operators behave in this way. For example:

```
int a = 0,b = 0;
b = a+ =1;
```

sets `a` and `b` to `1`.

You can even go as far as:

```
int a = 1,b = 2;
b+ = a+ =3;
```

This is evaluated as `a` is set to the contents of `a` plus `3` with the assignment returning `4` as its result. Then `b` is set to the contents of `b` plus the expression to the right, i.e. `4`. As a result `a` is set to `4` and `b` is set to `6`.

You can see that this is the same as:

```
b = a+b = a+3;
```

but of course you can't use this form as you cannot assign a value to `a+b`.

It is probably better not to use such confusing constructs.

The comma

The comma operator is perhaps the most puzzling in C. The comma is only treated as an operator if it is part of an expression:

```
expression1, expression2;
```

First `expression1` is evaluated and the result is thrown away. Then `expression2` is evaluated and it is returned as the result. This is very similar to the way assignment works but there are no side effects.

For example:

```
int ans = 1+2, 3+4;
```

this evaluates `1+2` and ignores the result and then evaluates `3+4` and stores this in `ans`.

The big problem with the comma is not what it does, but working out what you need it for. The obvious answer is that you want it when an expression has a side effect that you are interested in more than its result.

For example:

```
int ans= i++,a+i;
```

first increments `i` and then stores `a+i` in `ans`.

You can add the comma to the assignment so as to set up the initial conditions for a conditional. That is:

```
while (setup, (a = expression)<0){};
```

where *`setup`* is some action needed to evaluate the expression correctly. For a simple, if bizarre, example:

```
int a = 1,b = 2;
while (printf("%d\n",b), (a = b)<10){
  b++;
};
```

In this case `printf` is called and as `printf` is a function this is therefore an expression. Then `b` is assigned to `a` and the test of the value of `b` is carried out.

There is no point in writing this in this way as the `printf` might as well be in the body of the loop. Typically you will find the comma operator used where a function has to be called to initialize something before testing its current state. As before this is not a recommended construct, but you will find it in use.

The Ternary Operator

The ternary operator is often introduced as a shortcut to an `if ... else` construct. It is easy enough to understand:

```
(condition) ? TrueExpression : FalseExpression
```

If the condition is true then the expression evaluates *`TrueExpression`*, otherwise *`FalseExpression`* is evaluated. This can be thought of an expression equivalent of:

```
if(condition){
   variable = TrueExpression
}else{
   variable = FalseExpession
}
```

but notice that the if statement has to use a variable to assign the result to, that is the if statement is not an expression. This means you can use a ternary operator anywhere you can use a simple value. For example:

```
printf("%d\n",(ans>0) ? ans : -ans);
```

which prints `ans` as a positive quantity even if it is negative.

You obviously cannot include any statements in a ternary operator, i.e. no loops or conditionals, but this is not a real restriction as you can always define a function which includes statements and use that in the expression.

Used carefully, the ternary operator can make the logic of a program easier to follow. Used too much and it renders the logic impossible to follow.

The !! Operation

The way that C treats logical operations is slightly strange. Any zero value is considered as `False` and any non-zero value is considered as `True`. This is reasonable, but if you use any logic operators the result is always `0` or `1`. For example:

```
int c = 42||43;
```

sets `c` to `1`. In many languages that use the same loose definition of `True` and `False`, `c` would be set to `42` – still a representation of `True`. The difference is that C "normalizes" general values to `0` and `1` as the result of logical operations where other languages don't. The `0/1` representation of `False/True` was formalized in C99 with the introduction of the `_Bool` data type. The header `stdbool.h` also provides `bool` as a user-friendly alias for `_Bool` and the macros `true` and `false`. If you want to use `true` and `false` in earlier versions of C you can define your own type and constants:

```
typedef int bool;
#define TRUE  1
#define FALSE 0
```

Notice that logical expressions also use auto promotion. For example:

```
42||43;
```

can be thought of as:

```
(int)((int)42||(int)43);
```

Now we can examine the `!!myVar` operation. This is just the application of the NOT operator twice. The first time it negates the variable which produces `0` if `myVar` is non-zero and `1` if it is zero. The second application negates this result and so we have `!!myVar` is `0` if `myVar` is zero and `1` otherwise.

In other words, `!!` normalizes the value to `0` or `1`, i.e. false or true and sometimes this matters.

The Downto --> Operation

The very first thing to say is that there is no downto operator – it's a strange and misleading use of two standard operators. For example:

```
int x = 10;
while (x--> 0)
{
    printf("%d\n", x);
}
```

This is fancifully read as "while x goes to zero" and it does print `0` to `9` but not because of the use of the downto operator. The condition in the loop:

`x--> 0`

is actually `(x--) > 0` and the value of `x` is decremented and then tested against `0`. The illusion is courtesy of C's operator precedence rules that allow you to avoid using brackets.

Dive 5

Bits and More Bits

All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value.

Carl Sagan

It really is all about bits and nothing matters more to the C programmer. When using other languages you can abstract away so far from reality that you can forget that bits exist at all. Of course, the fact that you choose C means that this isn't what you want to do. If you are programming hardware or interacting with the operating system, individual bits are often the focus of your attention and working with them brings us to the subject of bit manipulation.

Addressing the Bit

You might think that there would be a direct way of getting at bits, but there isn't. The reason is simply that hardware generally accesses a group of bits at a time. The memory is divided into locations, each of which corresponds to a fixed number of bits at a specific address. That is, when you supply a memory address you specify a group of bits – an address rarely specifies a particular bit. You could build a single-bit-wide computer, but it would be very slow and not particularly practical. For reasons of history, the smallest addressable group of bits is generally a byte, i.e. 8 bits, but again there is no theoretical reason for this, it is just a reasonable number to work with. The most commonly encountered hardware, x86/64 and ARM, use byte addressing, but they may well actually work with the memory in larger chunks. A commonly used nomenclature is:

byte	= 8 bits
word	= 2 bytes = 16 bits
double word	= 4 bytes = 32 bits
quad word	= 8 bytes = 64 bits

but this use isn't universal and isn't the subject of a standard so you always need to check in any given situation.

The whole question of how addressing actually works and what is efficient is very dependent on the hardware in use. The implementation of a memory cache also complicates things in that typically 8-byte, double words, are read and written to memory irrespective of the addressing mode in operation. In other words, you might think you are reading or writing a byte, word or double word but the hardware will do what it does to be efficient and does its best to keep this hidden from you.

What all this means is that you cannot reason in the abstract about what might be the most efficient way to read some bits. There may be no simple answer to which is faster – byte access, word access, double word access or quad word access – because the memory architecture of the machine may not be as important as the size and type of cache in use. For example, in most cases reading bytes sequentially will be faster than reading bytes that are separated in memory by eight or sixteen bytes. The reason is that sequential bytes generally can be read from the cache without having to reload, but well-spaced reads might trigger a cache reload on each operation. Even this isn't certain because it depends on how full the cache is and how much free space there is.

If you are concerned about speed of access to a group of bytes, the only way to find out is to run a benchmark. You cannot deduce the performance from simple consideration of the addressing mode. This said, there is often a need to align an address with a word or double word boundary. For example if the address starts at `0` then the next byte is at address `1`, the next word is at `2`, the next double word is at `4` and the next quad word is at `8` and in general words start on addresses divisible by `2`, double words divisible by `4` and quad words divisible by 8.

The Bitwise Operators

C has a number of operators designed to allow you to perform bit manipulation. There are four bitwise operators:

AND	`&`
OR	`|`
XOR (exclusive or)	`^`
NOT	`~`

As you would expect, the NOT operator has the highest priority.

Notice that there are also corresponding Boolean operators, `&&`, `||` and `!` which only work with Boolean values – with zero as false and anything non-zero as true - and not with bit patterns.

The bitwise operators work with integer types. For example:

```
int a = 0xF0;
int b = 0xFF;
int c = ~a & b;
printf("%X\n",c)
```

This first works out the bitwise NOT of `a`, i.e. `0F`. This is then bitwise ANDed with `b`, i.e. `0F & FF` which is `F`. The `%X` format specifier prints the value in hex. You can use `%x` for lower case and `%d` for decimal.

Signed v Unsigned

Now we come to a subtle and troublesome point. Bitwise operators are only uniquely defined for unsigned values. The reason is that unsigned values have an unambiguous representation in binary and hence the operations are well-defined in terms of the values the bit patterns represent.

That is:

```
unsigned int value=5;
```

is always `0101` and hence:

```
value | 0x2
```

is not only always `0111`, but also always represents `+7`.

The same is not true for signed values simply because the way in which negative numbers are represented isn't fixed. The most common representation is two's complement, but this is not part of the C language standard, before C23, and so logical operations on unsigned numbers are implementation-dependent. The C23 standard changes this and makes two's-complement mandatory for negative values, but it will be some time before this comes widely into effect. Notice that this does not mean logical operations on signed values are undefined behavior – if this was the case most C programs would stop working properly. In addition "implementation-dependent" has one fairly consistent meaning:

> *take the bit pattern that represents the value and perform the specified logical operation as if everything involved was an unsigned value.*

This is the only sane way to deal with the problem and it is exactly what you would expect.

For example:

```
signed int value = -5;
```

if the machine uses two's-complement representation then the bit pattern is:

```
11111111111111111111111111111011
```

and:

```
value | 0x4
```

changes the third bit to `1`, i.e. `-1` in two's-complement.

As long as you assume a specific representation for the signed value then the logical operation is usually well defined as the bitwise application of the operator on the bit pattern. It is the representation that is implementation-dependent and not the operation.

Masks

Bitwise operations are the key to accessing individual bits. We usually think of this as setting or unsetting a particular bit or testing its state. You can set and unset bits using another value, usually called a mask, that defines the bits to be changed. In this sense the bits in the mask act as the addresses of the bits that you want to work with. For example, if you only want to change the first (least significant) bit then the mask would be `0x01`. If you wanted to change the first and second bits the mask would be `0x03` and so on. That is, the mask has the bits you want to access set to one with all other bits zero.

The mask gives the address of the bits you want to work with and what you do to those bits depends on the operation you use to combine the mask with the value. For example:

```
value | mask;
```

returns a bit pattern with the same bits set to one as in the mask. Notice that the bits that the mask doesn't specify, i.e. are zero in the mask, are left at their original values.

If you `OR` the mask you set the addressed bits to one and leave all of the others unchanged. For example:

```
int mask = 0x03;
int value = 0xFFF0;
int result = value | mask;
printf("%X\n",result);
```

sets `result` to `0xFFF3`, i.e. it sets the first (least significant) two bits.

Similarly if you use:

```
value & ~mask;
```

then the bits specified in the mask are set to zero, or are “unset” if you prefer this jargon. Notice that you have to apply a NOT operator to the mask. For example:

```
int mask = 0x03;
int value = 0xFFFF;
int result = value & ~mask;
printf("%X\n",result);
```

sets result to `0xFFFC`, i.e. it unsets the first two bits.

As well as setting and unsetting particular bits, you might also want to "flip" the specified bits, i.e. negate them so that if the bit was a one it is changed to a zero and vice versa. You can do this using the XOR, exclusive or, operator:

```
value ^ mask
```

which flips the bits specified by the mask. For example:

```
int mask = 0x03;
int value = 0xFFFF;
int result = value ^ mask;
printf("%X\n",result);
```

sets `result` to `0xFFFC` because it changes the lower two bits from ones to zeros. Again, bits not specified by `mask` are unaffected.

If you want to update the flag rather than derive a new result, you can use:

```
&=
|=
```

and:

```
^=
```

to perform the update directly. For example:

```
int value = 0xFFFF;
int value ^= 0x03 ;
printf("%X\n",value);
```

To summarize:

- Create a mask that has `1` at the position of each bit you want to change.
- To set the designated bits OR the mask with the value.
- To unset the designated bits AND the NOT of the mask with the value.
- To flip the designated bits XOR the mask with the value.

Bits in the mask that are `0` are unaffected by any of the operations.

A Single Function To Write Bits

Let's combine these operations into a single `bitWrite` operation that uses a mask to specify the bits to change and a data value to specify what each bit should be set to. If you regard the mask as the addresses of the bits to be changed then this writes the data to those bits and only those bits.

For example:

```
int mask = 0x03; //011
```

specifies that the first two bits should be modified and:

```
int data = 0x02; //010
```

specifies that the first bit should be set to zero and the second should be set to one.

The trick to working out how to do this is to construct one mask to set the bits that need to be set and another to unset the bits that need to be unset.

Then if a bit is to be set, it needs a `1` in the `mask` and a `1` in the `data` and the mask to set bits is:

```
setmask = mask & data
```

If a bit is to be unset it needs a `1` in the mask and `0` in the data, so the mask to reset bits is:

```
resetmask = mask & ~data
```

Applying both to the value gives the required result:

```
(value | setmask) & ~(resetmask) =
```

```
(value | (mask & data)) & ~(mask & ~ data)
```

which, after simplification, is:

```
value & ~mask | mask & data
```

Using this it is easy to create a function to do the job:

```
int bitWrite(int value, int mask, int data)
{
 value = value & ~mask | mask & data;
 return value;
}
```

For example:

```
printf("%b\n", bitWrite(0xFF, 0x0F, 0x05));
```

The mask means that only the first four bits are modified and the data sets them to `0101` which is the result of `11110101`.

You can use `bitWrite` to set any group of bits to any value.

Reading Bits

So far we have focused on writing bits, but we also often want to read a bit pattern and the method is more or less the same and relies on masks. As before, if you create a mask with bits set corresponding to the bits you want to read, then:

```
value & mask
```

returns `0` if and only if all of the bits the `mask` specifies are `0`.

Similarly:

```
~ value & mask
```

returns `0` if and only if all of the bits the `mask` specifies are `1`.

Usually, you only want to test for a single bit. For example:

```
int value = 0x1F;
int mask = 0x10;
int result = ~value & mask;
```

tests the fifth bit in `value` which is `1` and so `result` is `0`.

If you test for the fifth bit to be a zero using:

```
int value = 0x1F;
int mask = 0x10;
int result = value & mask;
```

then the result is non-zero, `16` to be precise.

If you want to convert the test results to Boolean values you can use the logical operator `!`. For example:

```
!(~value & mask)
```

is `true` if and only if all of the bits in `value` specified by `mask` are `1` and `false` otherwise.

Similarly:

```
!(value & mask)
```

is true if and only if all of the bits in the value specified by `mask` are `0` and false otherwise.

If you want to just read the bits specified by the mask, simply AND it with the value:

```
value & mask
```

For example, to read the bottom four bits you can use:

```
result = value & 0x0F;
```

Shifting Values

A very common requirement when reading bits is to move them to a suitable location within the group of bits. To do this you need the shift operators.

The << operator shifts the pattern of bits to the left, shifting zero into the low-order bit. So, for example:

```
int data = 0x0F;
int result = data << 4;
```

shifts the bit pattern in `data` four places to the left and so `result` contains `0xF0`.

Don't try specifying a negative shift as this is undefined behavior.

Similarly the >> operator shifts to the right, but there is a small complication concerning what is shifted into the new bit positions. What happens depends on whether the value being shifted is signed or unsigned. For an unsigned value `0` is shifted into the highest-order bit. For a signed value the same bit is shifted in as the one moved. That is, if the highest-order bit was a `1` then a `1` is shifted into the vacant position and it if was a `0`, a `0` is shifted in.

You can see that in the signed case the motivation with a right shift is to maintain the sign. That is, a shift keeps a positive number positive and a negative number negative. This sort of shift is usually called an arithmetic shift as distinct from a logical shift that always shifts a `0` in. You can always force a logical shift by casting the value to an unsigned type.

Notice that a left shift always shifts a `0` in, but this can change the sign of the value as a new bit becomes the sign bit. For example:

```
int data = 0x0F;
int result = data >>3;
```

shifts the bit pattern in data three places to the right and so `result` contains `1` since `1111` shifted three places right is `0001`.

For integers shifting one place to the left is the same as multiplying the value by two and shifting one place to the right is the same as integer division by two, but only for positive values. For example:

```
int data = 1;
int result = data <<1;
```

stores `2` in `result` and:

```
int data = 8;
int result = data >>1;
```

stores `4` in `result`.

Things are slightly more complicated for negative values. If you try:

```
int data = 7;
int result = data >>1;
```

then `result` is `3` which is what you would get with integer division, i.e. 3.5 is truncated or rounded down to 3. However, if you try:

```
int data = -7;
int result = data >>1;
```

you will find that result is `-4`. This is not what you would get with integer division, which would give `-3` after truncation. Right shift divides by two, but with rounding towards the next smallest negative integer. This is a subtle point that often causes errors.

The reason for this behavior is the way the sign bit is extended. For example:

```
int data = -1;
int result = data >>1;
```

stores `-1` in `result`. The reason is that the initial bit pattern is all ones, i.e. 32 bits all set to one, and so shifting one place to the right shifts in another `1` as the high bit. That is, `-1` is the same as `0xFFFFFFFF` and the sign bit is `1` so a shift right moves a one into the high-order bit, giving `-1` again.

Compare this to:

```
unsigned int data = -1;
int result = data >>1;
```

In this case a zero is shifted into the high bit and the value stored in `result` gives zero followed by 31 ones, which is a positive value equal to `2147483647`. Notice that the `-1` is stored in `data` as `0xFFFFFFFF`, but this is interpreted as a positive value when stored in an `unsigned int`. The result of the shift is `0x7FFFFFFF`, which is a positive number when assigned to a `signed int`. What this means is that you can use either right and left shifts or multiplication and division by two to do almost the same job with a little care over negative values. Shifts are much faster than multiplication and division because the hardware generally supports a direct shift operation in a register.

If you want a logical right shift on a signed value then the simplest way of implementing it is to use a cast:

```
int data = -1;
int result = (unsigned int) data >>1;
```

You can also use `<<=` and `>>=` to shift the value of a variable in place. For example:

```
int data =-1;
data <<= 1
```

shifts the bit pattern in `data` one place to the left and stores the result in `data`, i.e. it shifts `data` one place to the left.

Significant Bits

When working with a group of bits as a bit pattern we still tend to refer to the most significant and least significant bits, even though these terms really only apply when you are interpreting the bit pattern as a number. The most significant bit is associated with the largest power of two in a place value system and the least significant with the smallest power. Conventionally we write the most significant bit on the far left, following the way we write place value numbers in decimal:

You can assign all of the bits in a bit pattern a number starting from the lowest-order bit. Can you use the bit position as an address? Yes, as it is trivial to generate a mask from a bit position `p`:

```
mask = 1 << p
```

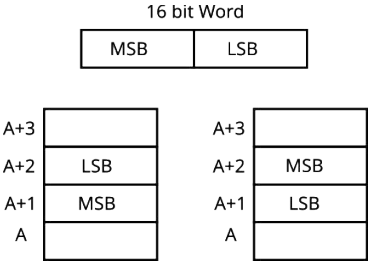
and if you want a mask corresponding to more than one position you can use:

```
mask = 1 << p1 | 1 << p2 | 1 << p3 …
```

This isn't a common requirement, but it can be useful when creating code that works with hardware registers that are often defined in terms of bit position.

Endianism

A problem that often arises in bit manipulation is endianism and it is closely connected to the ideas of bit position and byte addressing. When you design a computer there is a choice to be made. If parts of a word, say each byte, has a separate address do you store the low-order bits at low addresses, little-endian, or at high addresses, big-endian?



Although this is expressed as a problem that has to do with integers, it applies to any bit pattern that is broken up into bytes – what order do you reassemble it in. It also applies to bytes that are sent over any communication link – should the most significant byte be sent first or last?

Often you can ignore endianism as the machine takes care of it for you. If you save a 16-bit `int` then it will be saved in the order the machine uses and when you retrieve it the same order will be obeyed. Generally endianism only matters when the bytes have been acquired one-by-one from an external source such as hardware, a file or a communications link. Within a program, endianism only matters when you are aliasing a pointer or using a union, see Dive 15.

For example, consider a 2-byte `int` and alias it to a 2-byte array.

```
union endian{
    unsigned short myint;
    unsigned char mychar[2];
}test;
test.myint=0x00FF;
printf("%x \n",test.mychar[0]);
printf("%x \n",test.mychar[1]);
```

The `union` aliases `myint` and `mychar`. We then set `myint` so that all eight of its low-order bits are `1`. If the machine is little-endian, as in the case of x86 and ARM, then the low-order bits are stored in `mychar[0]`. If the machine is big-endian, e.g. Atmel AVR32, the low-order bits are stored in `mychar[1]`.

For example, creating a union to access the separate color channels ARGB – Alpha, Red, Green and Blue values in an unsigned 32-bit int is an example of something that changes according to endianism. As A is the high-order byte and B the low-order you might expect the union to be:

```
union Pixel{
    struct {unsigned char a,r,g,b;};
    uint32_t value;
};
```

and this works for big-endian systems as `b` the least significant byte has the highest address in the union.

However the x86 and ARM are little-endian. So the ARGB bytes are stored in the opposite order with `a` at the lowest address in the union:

```
union Pixel{
    struct {unsigned char b,g,r,a;};
    uint32_t value;
};
```

This is also a problem when data is transmitted in big-endian format over most Internet protocols – IPv4, IPv6, UDP, TCP etc. What this means is that if you are using a machine with little-endian format you transmit `mychar[1]` before `mychar[0]` to send a 16-bit integer when using almost any Internet protocol. That is, it is the order that data is sent and received rather than addressing that is variable.

Notice that some machines support both big- and little-endian and you can select which is used. It is also possible for different data formats, e.g. integer and floating point, to use different orders. You will also find issues of which byte goes first when you use memory-mapped devices. In this case, a careful reading of the data sheet usually solves the problem. If you are getting strange results and are sending data processed in bytes then you need to consider endianism.

Rotate

There is another form of shift – the left or right rotate – and while most processors have an instruction that will perform this type of shift in a single operation, for reasons of history C doesn't have a rotate operator.

A right rotate shifts all of the bits to the right and uses the least significant bit that is shifted out as the new most significant bit. That is, the low-order bit is shifted into the high-order bit. You can see that this is rotation of the bits.

Similarly a left rotate shifts all of the bits left and moves the high-order bit to become the new low-order bit. Again, you can see that this rotates the bits in the other direction. Rotations don't correspond to simple multiplication or division and they aren't useful in the same way as left and right shifts are in isolating bits. What they are used for is in cryptography and signal processing.

C may not have a rotate operator, but it is easy to implement a rotate using left and right shifts and logical operators. For example, a right rotate is:

```
unsigned int data=0xF;
data= data>>1 | data<<31;
```

The idea is easy enough to understand. First we perform a right logical shift which leaves the most significant bit zeroed. Next we perform a left logical shift 31 times to move the lowest significant bit to become the highest significant bit. The two are ORed together to give the result `0x80000007`.

You can see the general principle. If you want a right rotate of `n` bits of a 32-bit int then use:

```
data = data>>n | data<<32-n;
```

Similarly, if you want a left rotate of `n` bits of a 32-bit value then use:

```
data = data<<n | data>>32-n;
```

Notice that `n` cannot be negative or equal to 32, which would trigger undefined behavior. Also notice that data has to be unsigned so as to use logical shifts.

It is said that most compilers will detect the above idioms and convert the instructions into a machine code rotate instruction if possible. GCC does this optimization, but only if you make sure that the rotation is positive. That is:

```
unsigned int data=0x0F;
data= data>> (unsigned) 1 | data<< (unsigned)31;
```

compiles to:

```
ror  %eax
```

where `ror` is a rotate right instruction. If you leave off the unsigned qualifiers the line compiles to the equivalent right shift and left shift ORed together.
It would be better if C had a rotate operator.

Bit Fields

There is an alternative to some bit manipulation – the bit field. A bit field is a way of breaking down the storage into even smaller units than bytes. You can define a bit field within a `struct` by following the field definition by a colon and the number of bits. For example:

```
struct Bits{
    int bit1:1;
    int bit2:1;
};
```

creates an `int`, say four bytes, but only the first two bits are used. This is somewhat like a `union`, but in this case the bit fields share the `int`. For example:

```
struct Bits bits;
bits.bit1=1;
bits.bit2=1;
printf("%d\n",bits);
```

If we make a small change to the `struct` then you can see that you can work with more than just single bits:

```
struct Bits{
    int byte:8;
    int bit1:1;
    int bit2:1;
}bits;
```

This divides up the int so that the first eight bits are called `byte`, followed by two single-bit fields. Now we can write:

```
bits.bit1=1;
bits.bit2=1;
bits.byte=255;
printf("%d\n",bits);
printf("%d\n",bits.byte);
```

you will discover that `bits` is `1023` and `bits.byte` is `-1`.

As the `int` being shared is a `signed int` then so are all of the fields including the single-bit fields:

```
printf("%d\n",bits.bit1);
```

prints `-1`.

The standards are unclear on whether a bit field `int` is to be interpreted as signed or unsigned.

You can change the interpretation of a single field using:

```
struct Bits{
    unsigned int byte:8;
    int bit1:1;
    int bit2:1;
}bits;
```

`bits.byte` now prints `255`, but `bits.bit1` still prints `-1`.

You can often use other types, not just `int`, but this goes beyond standard C. If you mix types then usually the largest type is used. For example:

```
struct Bits{
    unsigned char byte:8;
    char bit1:1;
    char bit2:1;
}bits;
```

usually allocates a `char` for the first eight bits and another `char` for the next two bits, giving an overall size of two bytes for the struct. It is usually better to keep all of the bit fields the same type within a struct.

In general no padding is used in a bit field, but if you allocate more bits than an int can represent, a second int is used. If you want to force a new allocation so that a bit field is aligned to an address boundary you can include an unnamed bit field of size zero. For example:

```
struct Bits{
    unsigned int byte:8;
    int :0;
    int bit1:1;
    int bit2:1;
}bits;
```

allocates two ints with `bits.byte` occupying the first eight bits of the first int and `bits.bit1` and `bits.bit2` at the start of the second byte.

You can form expressions with bit fields and any result will not overflow into an adjacent bit field i.e. the bit fields are isolated. For example:

```
bits.byte = 255;
bits.byte++;
```

results in `bits.byte` overflowing and being set to zero, but `bits.bit1` isn't changed.

Notice that you cannot have a pointer to a bit field, but you can have a pointer to the struct that has the bit fields. This is often useful when you want to map a bit field to a hardware port or register.

Bit fields seem like a great idea, but their big disadvantage is that different compilers implement bit fields in different ways. For these reasons many programmers recommend avoiding the use of bit fields and implementing the bit operations using logic.

Some Examples

Count the Bits

Bit manipulation has a fascination that will keep many a programmer occupied for hours when they could be doing something more useful. To see why it is addictive, consider the following simple problem – count the number of bits set to one in a bit pattern. The direct way of doing the job is to use shifting and test each bit that is shifted out.

```
unsigned value = 0x55;
int count = 0;
for(int i = 0;i < 16;i++){
    if(value & 0x01) count++;
    value>> = 1;
}
```

Simple and direct, but not the only way. The best known solution is due to Brian Kernighan:

```
unsigned value = 0x55;
int count = 0;
while(value){
    value &= value-1;
    count++;
}
```

Why does this work?

The reason it counts the one bits can be understood by working out what happens the first time through the loop. If `value` ends in a `1` subtracting one removes it, e.g `101 – 1 = 100` ANDed with the original gives `100`. If `value` ends in a `0` then subtracting one has to borrow from higher-order bits. In fact, it borrows from the first high-order bit that is a `1`. For example, `100-1` has to borrow from the third bit in the pattern and the result is `011`, which when ANDed with the `100` gives `000` and the loop ends.

You can see that each time the subtraction causes a borrow from the first `1` bit to the right and this removes it from `value` and the AND removes the lower-order bits that result from the subtraction – hence the loop repeats the number of times that there are one bits in the bit pattern.

If you don't follow, try it out for other bit patterns and marvel that anyone noticed that this works in general and provides a way to count bits.

This is very typical of creative bit manipulation.

Largest Power of 2 that Divides

Sometimes you need to know the largest power of 2 that can divide a value. This is simply the number of zeros before the first one in the binary representation. For example, 112 in decimal is 1110000 in binary and hence $2^4 = 16$ is the largest power of 2 that divides it and $16 \times 7 = 112$. Also notice that 16 in decimal is 0010000 in binary. That is the power of two we want to find is just the binary number with a single one in the position of the first one in the original value i.e. 1110000 → 0010000.

You might notice that this is very similar to the counting the number of ones in a value only now we want to count the number of zeros before the first one. A similar technique to the Kernighan bit counting method will work.

If we first find the one's compliment, see Dive 2, of the value, that is `~value` we now have a value that has ones where the original had zeros. Add one to this to form `~value+1` and the carry will ripple through all of the one bits until it reaches the first zero which it converts into a one.

For example:

```
~0111 0000 + 1 = 1000 1111 + 1 = 1001 0000
```

If we now AND the original value with the result `value & (~value+1)` we get the largest power of 2 that divides `value`.

```
1001 0000 & 0111 0000 =  0001 0000
```

We can simplify this expression as `(~value+1)` is just the two's complement of value which is of course just `-value` for a signed type. So we have:

largest power of 2 that divides `value` is `value & -value`.

Parity

Parity is `1` if the number of `1` bits is odd and `0` if it is even. There are direct methods of computing parity, but a modification of the bit counting trick gives a good method:

```
unsigned int value = 4;
unsigned int parity = 1;
while (value)
{
  parity = !parity;
  value = value & (value - 1);
}
```

If you follow the bit counting example this should be obvious.

Swapping Values

Usually, if you want to swap two values, you need a third variable to hold the result while the swap occurs:

```
int temp = a;
a = b;
b = temp;
```

You can swap without a third variable using XOR:

```
a = a^b;
b = b^a;
a = a^b;
```

If you try it out you will find that `a` and `b` have swapped values. Seeing why it works takes a little longer!

Overflow-Free Average

Consider finding the average of two integers. The simple straightforward way of doing the job is:

```
int average = (a+b)/2;
```

This has one problem – `a+b` can overflow even if the answer is representable in the number of bits available. At first it looks as if nothing can be done, but if a result can be represented then there is usually a bit-manipulation way of computing it without the threat of overflow.

In this case we need to notice that addition is a simple logical operation. If you add the two values, `a` and `b` together then the sum bits are:

```
a ^ b
```

and the carry bits are:

```
a & b
```

this is how a half-adder is implemented.

For example:

```
a      1 0 1
b      0 1 1
sum    1 1 0
carry  0 0 1
```

To get the final result you have to add the carry bits to the next highest bit of the result. So a left shift and an addition:

```
  1 1 0 +   (0 0 1  <<1)
```

gives `1 0 0 0`

That is:

`a+b = (a^b)+(a&b)<<1`

and so:

`(a+b)/2 = !(a^b)>>1 + (a&b)`

which can be computed with no risk of overflow.

Dive 6

The Brilliant But Evil for

"We all know Linux is great...it does infinite loops in 5 seconds."
Linus Torvalds

If you need proof that C is a machine independent assembler, look no further than the `for` loop. It is closer to the metal than a for each or for in loop found in other languages and yet it has been deeply influential. So much so that the C for loop has made its way into Java, JavaScript, Go and many other higher-level languages.

The C for loop is brilliant in that it allows many "creative" uses of the construct and it is evil for exactly the same reason. More accurately it fails to capture the fundamental idea of an enumeration.

Quick Guide To The Loop Zoo

In the early days of programming we created loops to repeat instructions. This was a largely ad-hoc affair and not many stepped back from the code face to consider what was going on. Later we could consider the whole idea of repeating something and how many different varieties of loop there actually are and are actually needed.

The first big distinction is between the enumeration and the conditional loop. An enumeration loop is simply doing something a given number of times. Sometimes the number of times is implied by the number of things to process, e.g. the number of items in a data structure that need examining. Pure enumeration loops are often written as for loops of varying abstraction:

```
for i=1 to 10
for each x in array
for each value in 1 to 10
```

and so on.

Some languages even have enumeration loops like:

```
repeat 10 times
```

although this form isn't particularly flexible.

The alternative to an enumeration loop is the conditional loop which repeats something until a condition is satisfied. You can tell the difference between the two loops by asking the simple question “do you know how many times the loop will repeat before the loop starts?”. If you know that the loop will repeat x times before it even starts then you have an enumeration loop.

Even when you do have an enumeration loop it is sometimes easier to write it as a conditional loop.

Conditional Loops

A conditional loop has one of a number of possible forms but a common one is:

```
initial set up
test to see if loop should continue
   body of loop
   update test condition
end of loop
```

For example using C to implement this sort of loop directly:

```
    int i = 0;             ← initial set up
loopStart:
    if (i >= 10)           ← test to see if loop should continue
        goto loopEnd;
    printf("%d\n", i);     ← body of loop
    i++;                   ← update test condition
    goto loopStart;
loopEnd:
    printf("end of loop"); ← end of loop
```

You can clearly see the different parts of the loop and the way that gotos are used to jump to the appropriate points in the code. Not long ago all programming was like this and it mirrors the way that a conditional loop is implemented in assembler. Today we prefer to use high-level language constructs that bundle all of the sections up into fixed parts of the construct so you don’t have to waste time building your own from scratch.

The C for Loop

The conditional loop in the previous section is in fact an enumeration loop and could more easily be expressed in Basic as:

```
for i = 0 to 9
   print(i)
next i
```

or in Python

```
for i in range(0,10):
    print(i)
```

C takes the approach that it is better to package the parts of a general conditional loop rather than invent a proper enumeration loop. The C for loop, and it would be better called a conditional loop, is:

```
for(initial set up; test if loop should continue;
                                        update test condition){
   body of loop
}
```

When you write a C for loop the compiler uses the expressions to construct a conditional loop. For example:

```
for(i = 0; i < 10; i++){
   printf("%d\n", i);
}
```

is equivalent to the previous conditional loop:

```
   int i = 0;
loopStart:
    if (i >= 10)  goto loopEnd;
    printf("%d\n", i);
    i++;
    goto loopStart;
loopEnd:
    printf("end of loop");
```

The `i = 0` is performed before the loop starts, the `i < 10` is evaluated at the start of the loop and the body of the loop only executes if it is true and the `i++` is performed at the end of the loop before the next potential repeat. Notice that the condition in the for loop has to be true for the loop to continue. In the example conditional loop the condition has to be true for the loop to end but this is a minor difference.

In this form the C for loop is perfectly OK as an enumeration loop. All C programmers learn this form as an idiom and read it as a loop that repeats for i from 0 to 9. It is as if the C for loop is frozen in a restricted but useful form:

```
for(index=startvalue; index<endvalue; index++){
```

and the only things you ever change are the *`startvalue`* and *`endvalue`* and the loop repeats for index from the *`startvalue`* to one less than the *`endvalue`*. In this form it really is an easy to use enumeration loop.

From here most C programmers learn to vary it slightly to get different effects.

For example:

```
for(index = startvalue; index <= endvalue; index++){
```

gives an enumeration loop that runs from *startvalue* to, and including, *endvalue*, whereas:

```
for(index = startvalue; index >= endvalue; index--){
```

gives an enumeration loop that runs from *startvalue* down to *endvalue* assuming *startvalue* >*endvalue* and so on.

These idioms are what makes the C for loop usable for beginners and non-beginners alike. It is a for loop with direct mapping to the underlying assembly language that it is compiled to and as such it is brilliant. But it has its evil side – it is too flexible.

While and Until

The for loop is very general, but it is limited to having its exit point at the start of the loop. This makes it a while loop.

Loops differ in where they place their exit points, i.e. where the test for the end of the loop is. This is another distinguishing feature in the loop zoo – loops differ in both the number of exit points they have and where they are placed.

While loops test at the start and until loops test at the end.

It should be clear that the difference is that a while loop can exit without ever executing the body of the loop, but an until loop has to execute the body of the loop before making a test. You can summarize this as a while loop repeats `0` to *n* times and an until loop repeats `1` to *n* times. In other words, an until loop has to execute at least once.

C has a while loop, but it is entirely equivalent to a particular form of the for loop. The while loop:

```
while(condition){
}
```

is exactly the same as:

```
for(;condition;){
}
```

and, yes, you can leave out any part of the for loop as long as it makes sense. Indeed:

```
for(;;)
```

is an infinite loop, often used in IoT applications and in constructing more general loops. It never ends because there is no condition specified.

C also has an until loop and this is one that is more difficult to convert into a for loop:

```
do{
  body of loop
}while(condition)
```

If this is an until loop, why does it end with `while`?

The answer is that in a conventional `until` loop the condition is for the loop to end. In a conventional `while` loop the condition is for it to continue. You can see that `until = !while` in that until ends the loop and while continues it.

So even though the C until loop has its exit point at the end it uses a condition for it to continue – hence the use of `while`. It's a historical accident caused by the desire to reuse the compiler's handling of the `while` condition i.e. to avoid introducing an `until(`*`condition`*`)`.

You can convert this to a `for` loop, but only if you are prepared to repeat the body of the loop:

```
body of loop
for(condition){
  body of loop;
}
```

Break and Continue

What about loops that have their exit point not at the end and not at the beginning?

Before languages like C, loops could exit from any point in their body simply by using a `jmp` instruction to outside the loop. This was not a good idea because it relied on the programmer to keep things simple and clear. For a long time it was asserted that you only needed a for loop, a while loop and an until loop – and this is true. Any program that can be written, can be written using just these loops. This is the original premise of "structured programming", the first attempt to make programming better.

Today we tend not to be as hardline and the `break` and `continue` statements were introduced in C90. The `break` statement is equivalent to a `goto` that jumps out of the loop, i.e. to the first instruction following the loop. The `continue` statement is equivalent to a `goto` that jumps to the start of the loop, i.e. it skips any remaining statements in the body of the loop.

Both constructs let you construct loops that complete a fractional number of repeats. For example:

```
for(i = 0; i < 10; i++){
   print("before\n");
   if(i == 5) break:
   printf("after\n", i);
}
```

This loop executes the `before` 5 times, but the `after` 4 times and you can see that there is a sense in which it repeats the loop body four and a half times.

You can write a completely general loop with any number of exit points anywhere in the loop using:

```
for(;;){
 first part of loop
   if(condition) break;
 second part of loop
   if(condition) break;
 third part of loop
    and so on
}
```

Find It and Fix It

If you are wondering what the `break` and `continue` statements are for, you have never encountered, or never thought about, the find it or fix it algorithms.

The find it algorithm is simply having to scan through an array to find a specified value. Many attempt to implement this as an enumeration loop because you know the length of the array `n`:

```
for(i = 0; i < n; i++){
       if(array[i] == target) hit = i;
}
printf("%d\n",hit);
```

The problem is that the loop repeats `n` times, even if the target is in the first element. You can fix this inefficiency using the `break` statement:

```
for(i = 0; i < n; i++){
       if(array[i] == target) break;
}
printf("%d\n",i);
```

This brings the loop to an end when the target is found but notice that it relies on the value of `i` not changing due to the operation of the `break`.

This is the case unless you use the C99 idiom of declaring the index in the `for` statement:

```
for(int i = 0; i < n; i++){
       if(array[i] == target) break;
}
printf("%d\n",i);
```

This doesn't work because `i` no longer exists at the end of the loop. It is also worth reminding the reader that a for loop is a block and as such variables declared within it only live for as long as the for loop is executing, see Dive 8.

In this case, however, the use of an enumeration loop isn't really justified. For a true enumeration loop you have to know the number of repeats before the loop begins. In this case we only know the maximum number of repeats is the number of elements in the array – the loop may finish earlier if the target is found. This is not an enumeration loop. It is a conditional loop where we know the maximum number of repeats and it can be written more expressively as:

```
i = 0;
while(array[i]! = target && i<n ){
   i++;
}
```

You can even write this loop as a C for loop:

```
for(i = 0; array[i]! = target && i<n; i++){
}
printf("%d\n",i);
```

However, many programmers think that having a compound condition to keep a loop running is less clear than a misuse of a for loop at the cost of a break.

Finally what is the `continue` for?

The `continue` is for the "fix it" loop. If you need to scan through an array and pick out elements that need to be "fixed" or processed in some way you might use:

```
for(i = 0; i < n; i++){
       if(array[i]! = fixFlag) continue;
       fix(array[i];
}
```

If the array element needs fixing then it is fixed, otherwise the `continue` skips the `fix` and moves the loop on to the next element of the array.

This is an enumeration loop proper, as we know the number of repeats before the loop begins. However, we don't really need the `continue` and can write instead:

```
for(i = 0; i < n; i++){
       if(array[i] == fixFlag) fix(array[i];
}
```

The `continue` just gives you a way of skipping the last part of a loop and you can always do this with an `if` statement.

The Evil Parts

The `for` is a flexible construct capable of being used to implement for loops and a range of different conditional loops. As such it is not a nice simple straightforward enumeration loop and is capable of great misuse.

As each part of the `for` loop can be an expression, the possibilities are great. For example, you can use three functions to control the loop:

```
for(init();test();inc()){
```

and obviously `init` performs the initialization, `test` returns a Boolean to control the loop and `inc` is evaluated at the end of the loop. In this form it is clear that the `for` loop can do almost anything and implement almost any type of loop with an exit point at the start.

In most cases it isn't even necessary to resort to using functions to introduce statements into the mix. Often you can get away with just using expressions separated by a comma operator. For example:

```
for(i = 1,j = 9;j > 0;i++,j--){
    printf("%d %d ", i, j);
}
```

Notice that we initialize both `i` and `j` and that `i` increments and `j` decrements each time through the loop. The result is a loop in which `i` runs from 1 to 9 and `j` from 9 to 1. There are alternative, clearer, ways of writing this loop.

Final Thoughts

C `for` loops aren't really enumeration loops and probably shouldn't have "for" in their name. In practice, they encompass a wide range of loop types and as such they are dangerous from the point of view of code clarity. You will encounter many robust opinions that this form of loop or that form of loop is evil and should be avoided, and some are. However, what matters, and it matters in all code not just the for loop, is clarity of intent. If a for loop is a good expression of what you intend it to do, then it is a good for loop, no matter what anyone else says.

One way to ensure that this is the case, without having to spend too much time on evaluation, is to restrict yourself to only a small number of variations - `for(i = 0;i < n;i++)` is good.

Dive 7

Into the Void

I'll get me to a place more void.
William Shakespeare

You can't help having encountered the mysterious void data type. It sounds like something from a SciFi novel or film and, along with the semi-colon, accounts for a C program's intimidating look to the beginner. As time ticks on, however, we get used to the void and eventually hardly notice it. But why "void" and what exactly does it mean?

Why Void?

As far as I can determine the use of "void" as a type was introduced in Algol 68 and its language designers also seem to have liked related terms like "voiding". The word is defined by wiktionary as:

1. Containing nothing; empty; not occupied or filled.
2. Having no incumbent; unoccupied; said of offices etc.
3. Being without; destitute; devoid.
4. Not producing any effect; ineffectual; vain.
5. Of no legal force or effect, incapable of confirmation or ratification.
6. Containing no immaterial quality; destitute of mind or soul.
7. Of a function or method that does not return a value.

My guess is that it is the first definition that comes into everyone's mind when they read "void" in a program, but I actually think it is the fifth that explains why we use it.

The original idea was that a C function should define, not only the types of its parameters, but the type of its return value. So you might write

```
int sum(int a, int b){…
```

to mean that `sum` returns an `int`. This allows the compiler to check that the function really does return an `int` at compile time.

Essentially, if the function has a `return c;` instruction and `c` is confirmed to be an `int` then everything is fine.

This idea raises the issue of what to do when a function is a procedure?

Functions always return a result. Functions that don't return a result are more properly called “procedures” or “subroutines”. Other languages sometimes provide different syntax for a function, which returns a result, and procedures, which don’t. In the case of C, and many modern languages, we simply have functions and we need a way to show that no result is returned. If you think about:

```
int sum(int a, int b){…
```

as being a contract that `sum` will return something, which in this case is an `int` then you might think that a function that didn’t return anything broke its contract to be a function, i.e. the contract was null and void.

Well that’s my theory and it helps to see why `void` might be appropriate.

Void In C

We now appreciate the idea of

```
void sum(int a, int b){…
```

declaring a function that doesn’t return a result – indeed that must not return a result. If `sum` did return a result then you would see a compile error something like:

```
'void' function returning a value
```

Notice that in C you can choose to ignore any return value. What happens is that the value that is returned on the stack is simply thrown away. For example in:

```
int sum(int a, int b){
 return a+b;
}
sum(1,2);
```

you are throwing away the result and you won’t even see a warning in most cases as this is perfectly valid C. Contrast this with a function that returns `void` which has to be called in this way.

You might be familiar with `void` being used to turn functions into procedures, but what about functions that don’t accept parameters? It is common to see:

```
int sum(){
 return 1+2;
}
printf(“%d\n”,sum());
```

but this is mostly wrong. It doesn’t declare a function with no parameters. It declares a function with a fixed, but unknown, number of parameters.

With that definition you can still write:

```
printf("%d\n",sum(1,2));
```

and generate no compiler or runtime errors. Of course, with this declaration the function cannot access the parameters, but you can separate the declaration and implementation of the function:

```
int sum();
int main(int argc, char **argv)
{
 printf("%d\n",sum(1,2));
}

int sum(int a, int b){
 return a+b;
}
```

In this case the function is defined after the main program. So the first declaration is needed and it declares a function with no parameters – only, of course, it doesn't. It declares a function with any number of parameters which is why the program works – you might see a warning message from a helpful IDE or compiler. This is so confusing that C99 outlawed the use of *`function`*`()` at all and you have to use *`function`*`(void)`.

If you replace `int sum();` by `int sum(void);` in the previous program you will see an error message as the definition of `sum` has parameters and its declaration states that it doesn't have any.

To summarize:

- Use `void function(...)` to state that a function definitely doesn't return a result.
- Use `type function(void)` to state that a function definitely doesn't take any parameters.

See Dive 13 for more.

Void Expressions

A little known use of `void` is to create a void expression. Expressions are like functions in that they return a result. Usually, if you want to ignore the value of an expression then you can simply do that, for example:

```
2+3;
```

Such expressions are generally called "statement expressions". If you want to make it clear that an expression is throwing away its result you can cast the result to void. That is, the previous statement expression is equivalent to:

```
(void)(2+3);
```

The Void Type

Even in the days of Algol 68, the `void` keyword's meaning had expanded beyond just stating that a function didn't return a value – it had become a type. Rather than being a type that breaks the rules, as in "null and void", `void` is a type that is anything. This is a little confusing for the beginner attempting to make sense of `void` used in a function – does it mean it returns a `void` type? Only if nothing is the void type and this isn't a good conclusion to draw.

What does `void` being any type actually mean?

You can't declare a variable of type "anything" because, in C at least, all variables have to have a known type so that the compiler can allocate storage for them. When you use:

```
int myVar;
```

the compiler usually allocates four bytes to the variable, although this is machine-dependent. What would the compiler allocate if you wrote:

```
void myVar;
```

The only sense that a void type can exist is if we can allocate a sensible amount of storage for it. The solution is to use `void` pointers. As explained in Dive 10, a pointer is a variable that stores the address of another item of data. A pointer always takes the same amount of memory to store, even though it is system-dependent. Of course, what the pointer references can be anything, so:

```
void *myPointer;
```

creates a pointer that can reference anything.

How best to think of a void pointer?

At the lowest level you can think of a void pointer as referencing the start of a block of memory, but this view has a problem. To access parts of the block of memory you would need to do pointer arithmetic. For example, to access the 20th element of the block you might try:

```
myValue=*(myVoidPointer+20);
```

The idea is that the value stored at the address given by `myVoidPointer` plus `20` is to be retrieved and stored in `myValue.` The problem here is we don't know what the basic unit of allocation is. In other cases, pointer arithmetic works in units of the basic type being referenced.

For example:

```
int *myIntPointer;
. . .
myValue=*(myIntPointer+20);
```

In this case the addition is equivalent to `20*sizeof(int)` and we are retrieving the 20th integer stored in the block of memory as if the block was an array.

So what does `myVoidPointer+20` mean?

By analogy it has to be: `myVoidPointer+20*sizeof(void)`

and the only sensible value for `sizeof(void)` is `0`.

This doesn't work and from C99 onwards void pointer arithmetic is not allowed. Earlier standards were confused on the issue. Some use: `sizeof(void)=sizeof(char)`

which sort of makes practical sense, but if this is what you want you could simply cast the pointer to `char`:

```
(char*)myVoidPointer+20
```

So don't do `void` pointer arithmetic. It doesn't make any sense, even if your compiler allows it.

If you can't use a `void` pointer to manipulate memory, what use is it?

Its only use is when you don't know what the type of the data will be or when you need to change its type on the fly, as for example in:

```
void *get(void *p, int type, int index)
{
    switch (type)
    {
    case 1:
        return (int *)p + index;
    case 2:
        return (char *)p + index;
    }
    return NULL;
}
```

You can see that the type of the pointer passed is resolved at runtime using the `type` parameter. Without this approach you would need a `get` function for each type.

Notice that we are making use of the rule that any pointer type can be cast to a `void` pointer and this is done automatically. So when we call:

```
int array1[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *element = get(array1, 1, 5);
```

the `int array1` is automatically upcast to `void`. If you prefer you can make this explicit:

```
int *element = get((void*)array1, 1, 5);
```

In short a `void` pointer is never used to process data, only to pass it around when its type isn't determined at compile time.

Final Word

`void` is a useful construction, but it isn't just one thing. It is an indicator that a function doesn't return a result and it can be used to specify that a function takes no parameters. As a type it is useless for processing data, but excellent for passing data of unknown type to a function, or in any other situation where the data type isn't known until runtime.

Dive 8

Blocks, Stacks and Locals

A whole stack of memories never equal one little hope.
Charles M. Schulz

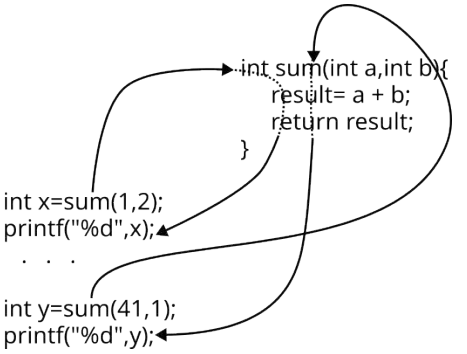
C is a block-structured language. It is also the language responsible for making nearly every other modern language block-structured and yet this idea is now so familiar that we miss exactly what this means. We also have almost forgotten why block-structured languages are so natural. The story all starts with the way functions, or originally subroutines, were implemented.

The Need To Return

All programmers are familiar with the idea of a function, also called a subroutine or procedure, but there was a time when they didn't exist. Early computers didn't have the capacity for such frivolous stuff. You only had enough memory to hold a program as it was written as one continuous block of code. Then programs became larger and the need to break them into blocks that did particular tasks became obvious and the subroutine was born. The basic requirement of a subroutine was the need to return to where it came from.

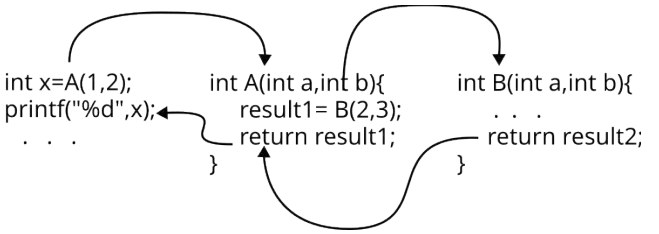
Today, in C, we call subroutines "functions", but everything else is the same. That is, code can "call" a function, expect it to do a job and then return with the results. This notion of "return" is so obvious to any programmer today we forget that it had to be invented and, more to the point, implemented.

Once you have a subroutine that works out the sum of two numbers you can reuse it over and over again and each time it will "return" to a correct, but different place:



The function call to `sum` takes the flow of control to the same place, but the `return` takes it back to the instruction following the call. In the early days, subroutines sometimes used a simple array to store the return address of the instruction following the call. Then things became slightly more complicated.

Functions can call other functions:



In this case the `main` program calls function `A` which calls function `B`. When `B` returns it goes back to `A` and then back to the `main`. That is, the functions are called in the order `main` → `A` → `B` but the return addresses are in the order `B`→ `A` → `main`. This reversal of order is typical of a Last In First Out (LIFO) stack.

For example, consider when `main` calls function `A`, which calls function `B` which calls function `C` which calls function `D`. That is:

`main → A → B → C → D`

What happens is that each return address is pushed on the stack as the function is called:

```
main → A              → B              → C            → D

push return main → push return A → push return B → push return C
```

This leaves the stack storing:

```
push return C
push return B
push return A
push return main
```

When function `D` finishes it pops the return address off the top of the stack, i.e. it returns `C`. When `C` finishes it pops its return address from the new top of the stack, i.e. it returns `B`

```
push return B
push return A
push return main
```

and so on.

You can see that if we follow the rule "push the return address on the stack when the function is called and pop the return address off the top of the stack when the function is finished" everything just works.

A stack follows the natural call and return flow of the function.

When this was noticed the stack became the standard way to implement functions.

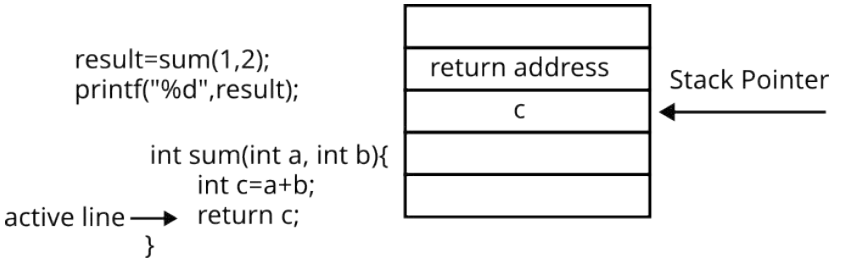
Today we use a stack to implement functions, but before this the return address was handled in an ad-hoc way. Often, a single location was used to store the return address and this resulted in the restriction that you could only call a function which in turn could not call a subsequent function. Function calls were restricted to a depth of one. Later additional return variables were added, but you were still restricted in the depth of function calls you could implement and it wasn't particularly efficient.

Automatic Local Variables

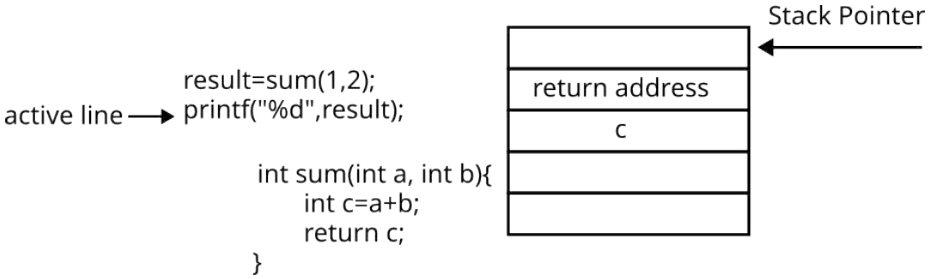
For some time, the stack was used just for the return address of functions. In those early days any variables that you created within a function had the same status as variables created in the "main" program. That is, when you used a variable in a function it became a top-level global variable which could be accessed from everywhere in the program. This made getting results out of a function easy, but it also made programming very error-prone because you couldn't write a function in isolation – you had to avoid variable names that were used elsewhere in the program.

Then someone noticed that the stack could be used to create variables that were automatically local to the function. This approach was made clear in the 1950s as part of the Algol language, but it took longer for other languages to adopt it.

The idea was simple. If a variable is declared within a function then the storage for it is allocated on the stack. While the function is active the variable is accessible and usable, we say it is “in scope”.



When the function ends the stack is popped back to the return address and this is used as the target of the return. The variables that were created on the stack are no longer accessible as the stack pointer has moved back to its starting position.



This approach allows languages, C included, to have variables declared in functions to be automatically local. Although you don’t see it used very often, basically because it doesn’t actually change anything, C has the `auto` keyword which defines a local variable. You can write:

```
int sum(int a, int b){
   auto int c=a+b;
   return c;
}
```

but the use of `auto` changes nothing. The variable `c` is an automatic local variable as it is defined within a function.

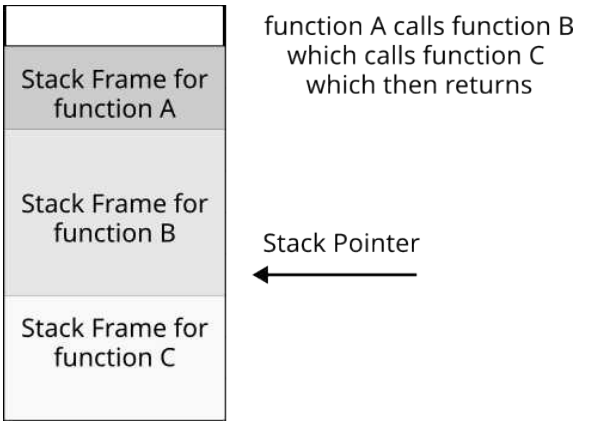
To be more precise about the way that the stack is used, we need to introduce some new jargon, “stack frame”. A stack frame is what is created on the stack when you call a function. The exact format of a stack frame varies according to the machine architecture and what optimizations the compiler is applying. All that really matters is that the information used by the function is stored in its stack frame on the stack and when the function ends the stack pointer is adjusted to reference the previous function’s stack frame.

Most modern languages use the stack and the stack frame to store the return address, all the local variables and the functions' parameters. That is, when you call a function, the arguments you supply are usually pushed onto the stack to be retrieved and used by the function you call. In the same way, it is usual for any of the function's results to be pushed onto the stack as the function comes to an end so that the calling function can retrieve it.

Function call and return really fit neatly into the way a stack operates.

Dead Data

The operation of the stack during function calls and returns is fairly clear now, but there is a potential problem. For example, if function `A` calls function `B` then there are two stack frames on the stack. If function `B` then calls function `C` another frame is pushed onto the stack. If function `C` returns, using the return address stored in its stack frame then the stack pointer is set to the start of the stack frame for function `B`.



If you follow this idea you can see that it results in a perfectly natural way to implement functions, including their return addresses and automatic local variables.

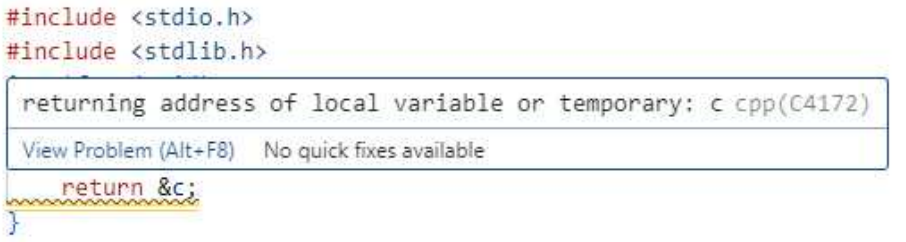
Notice that in the diagram the data that function `C` used is still stored on the stack because it isn't accessible using standard stack operations. What this means is that it is possible to arrange to access a function's local data after it has returned!

For example:

```
#include <stdio.h>
#include <stdlib.h>
int *funA(void)
{
    int c = 42;
    return &c;
}

int main(int argc, char **argv)
{
    int *result=funA();
    printf("%d\n",*result);
    return (EXIT_SUCCESS);
}
```

In this program `funA` creates a local variable `c` and stores `42` in it. It then returns the address of `c` and the main program uses this to print the value stored in `c`. In most cases this program will cause the compiler or the IDE to flag the potential error:



The warning message is perfectly correct in that `c` is a local variable and hence only "lives" while the function is executing, and as such you should not return its address. However, if you run the program there is a good chance you will see `42` printed because the stack where the value of `c` was stored hasn't changed. On some systems you will see a runtime error because the stack has changed enough to cause a memory fault. If another function is called after `funA` and before the value of `c` is printed then the chances are that you will not see `42`, but some other value as stored on the stack by the other function.

This may seem like a contrived example, it is, but it is a far more common error than you might imagine as we will discover in Dive 13: First Class Functions.

The important point is that once a function returns you have to regard all of its local variables and everything it stored on the stack as being destroyed, even though they may hang around for a while simply because some other function hasn't made use of the memory as yet.

Blocks

So far we have focused on functions, but C is a block-structured language and functions are just one example of a block. The basic idea is that a block is a collection of instructions contained by curly brackets.

`{`*`statement1`*`;`*`statement2`*`; ...}`

This is often called a compound instruction to emphasize that it behaves as if the list of instructions it contains acts like a single big instruction. For example, the C flow of control statements like `if` are only defined for a single instruction:

`if(`*`condition`*`)` *`statement`*`;`

and the statement is obeyed only if the condition is true. If you want to execute multiple statements then you need to use a block:

`if(`*`condition`*`) {`*`statement1`*`;`*`statement2`*`; …}`

This much all C programmers know, but a block is more than just a compound statement. A block acts like a function in the way it deals with the stack. That is, when you enter a block, any new local variables are added to the stack frame and when you leave the block they are lost. Another way to think of this is that a block is like a function, but without parameters and a `return`. Or you could say it the other way round - a function is a block with parameters and a `return`.

For example:

```
int main(int argc, char **argv)
{
    int a = 42;
    {
        int c = 43;
        printf("%d   %d\n", a, c);
    }
    printf("%d\n", a);
    return (EXIT_SUCCESS);
}
```

In this case the block introduces the new variable `c` and this is only accessible within the block. If you try to use `c` outside of the block then the compiler will signal an error that you are trying to use an undefined variable. Before C99 all of the declarations in a block had to be at the start of the block, but now they can be anywhere. Notice that `a` is accessible inside the block this is usually called lexical or static scope.

If you know JavaScript, also notice that C does not implement hoisting for variable declarations. In JavaScript all declarations are moved to the start of a block, but in C the variable only exists after it has been declared.

Things are a little more subtle than this. If the variable is already defined outside of a block then any redefinition inside the block "shadows" the original variable, which is unavailable until the block is exited.

For example:

```
int main(int argc, char **argv)
{
    int c = 42;
    {
        int c = 43;
        printf("%d\n", c);
    }
    printf("%d\n", c);
    return (EXIT_SUCCESS);
}
```

You will see `43` printed then `42` as the two different incarnations of `c` come into scope. As C variables of the same name but different type are regarded as the same variable it doesn't matter if you declare the inner variable to be of a different type:

```
int main(int argc, char **argv)
{
 int c = 42;
 {
   float c = 43;
   printf("%f\n", c);
 }
 printf("%d\n", c);
 return (EXIT_SUCCESS);
}
```

In this case the inner block's `c` still shadows the containing block's `c` even though they are different types.

Notice that you can nest one block inside another. When you do, the variables declared in an inner block are local to that inner block, which can still access variables declared in any containing block. Any variables that you declare within the block that have the same name as variables in the containing block shadow the containing block variables even if they differ in type.

Why might blocks be useful? In fact you make use of them all of the time, perhaps without realizing. For example, as already mentioned, all of the C control statements make use of compound instructions and as such each is a block:

```
for(int i = 0;i < 10;i++){
    int c = 0;
    c = c+i;
    printf("%d\n",c);
}
```

In this case `c` is local to the block and cannot be accessed outside of the `for` loop. In addition if there is a `c` already defined before the `for` loop then it is unchanged after the `for` loop ends. Notice the `i` is also local to the block as C99 now allows declarations within code and the `for` is regarded as being within the block.

The same behavior is the case for all of the control statements `if`, `switch`, `while`, `for` and `do-while`. Any variables declared between curly brackets are local to the block and go out of scope when the block is exited.

Within control statements, the local scope of variables to the containing block is useful as it can allow the reuse of variable names. Occasionally it is useful to create a block that isn't part of a control statement within a program for exactly that reason. For example, if you have a section of code that is giving problems you can run it with slight variations by copying it and pasting it back into the program and surrounding it with curly brackets to be sure that it has no effect on other variables already in the program.

Final Word

The way that the stack works in exactly the way that fits with function call and return is pleasing. The way that it makes automatic local variables easy and natural is very surprising and not to make use of it would be to "look a gift horse in the mouth" – it is why local variables behave in the way that they do.

C takes this idea one stage further and allows you to declare blocks that are not functions. This is such a natural part of C programming you quickly forget that it is special and yet knowing exactly what is going on means you can be creative. If you are creative with blocks, remember to include some comments explaining what you are up to – share the creativity.

Dive 9

Static Storage

"Up until the 1920s, everyone thought the universe was essentially static and unchanging in time."

Stephen Hawking

We all know what a variable is, but often we don't. The simplest mental model of a variable is a memory location associated with a name – a label. You can think of the variable as a box that you can put a bit pattern into and retrieve by name.

Consider a simple assignment:

```
int a = 42;
```

You can think of this as creating a box called `a` and storing the bit pattern for `42` in it. Similarly, using `a` in an expression retrieves the bit pattern from the box. Of course, in practice the box is a memory location and the label is the location's address.

Sometimes it is important to know where the memory location is and how it is allocated. C supports two well-known ways of allocating memory. We have already met automatic variables, see Dive 8, which are allocated on the stack and are ideal for variables that have a block lifetime. Variables that are explicitly allocated on the heap, see Dive 11, are ideal for situations when you need the lifetime to be independent of any block. This is the stack versus heap distinction, but there are other ways that variables are implemented.

One of the most important is strangely named "file level storage" also known as "static storage".

File Level Storage

To understand how file-level storage works, we have to consider how a program is compiled and run. The compiler reads the C code and for each C instruction it outputs a number of lower-level machine code instructions. These instructions are simply bit patterns and only differ from data in that they are examples of machine code instructions and the processor can read and obey them.

That is, a program is just data you can execute.

The output of the compiler is a binary file and to run the program the loader simply loads the file into the correct location in memory and then makes the processor jump to the address of the first instruction.

The fact that the program file is just a binary file that is loaded into memory means that it is possible to include data within it for the program to work with. That is, memory locations within the program can be used to store data. Usually the compiler arranges to store all of the data together in one area of the file and so we have the idea of a data block and a code block. Depending on the system and the compiler there can be multiples of both types of block in a single binary file.

File-level variables are stored in the binary file that the compiler generates and are stored within the program. Thus they are neither automatic variables stored on the stack nor are they heap variables. A file-level variable is defined by being declared outside of a block, i.e. outside of a function. This means that the declaration is at the top level of the file of code and hence a “file-level variable”. For example:

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

int a = 42;

int main(int argc, char **argv)
{
    int b = 314159;
    printf("%d,%d", a, b);
    return 0;
}
```

The variable `a` is declared at the file level and variable `b` is an automatic local variable. Recall that in C the main “program” is just another function and variables declared within it are automatic and stored on the stack. Variable `a` is compiled into the data area of the program and isn’t allocated space on the stack or the heap – it is part of the program file. It is also initialized when the program file is loaded. As it is part of the program, `a` has the maximum lifetime – it lives until the program terminates and the

memory it occupied is reused by another program. This is very similar to the variable `b`, which lives for as long as the `main` function is active. In a single-threaded program life and main function life are the same as the program ends when the `main` function ends. In a multi-threaded program this isn't necessarily true.

A big difference between the two variables is that `a` is accessible, i.e. in scope, everywhere in the file. That is, all of the code in the same file can access `a`, whereas variable `b` can only be accessed from `main` even though it has the same lifetime as `a`.

For example:

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

int a = 42;

void myFunction(){
    printf("%d,%d\n", a, b);
}

int main(int argc, char **argv)
{
    int b = 314159;
    printf("%d,%d\n", a, b);
    myFunction();
    return 0;
}
```

This works and `myFunction` prints the value of `a`, but it will not print the value of `b`, even if you try to.

Another difference is the way file-level variables are initialized. Consider the instruction:

```
int a = 42;
```

In this case, as it is a file-level variable, it is allocated in the program file and that data is set to `42` in the file. When the program is loaded into memory the variable already has `42` stored in it and no further action is taken. Compare this to:

```
int b = 314159;
```

In this case the variable is an auto and so it is allocated on the stack when the function, `main`, is called, this means that the stack then has to be set to `42` as the program runs.

You can see that you get the initialization of the file-level variable for free as part of loading the program. This sometimes is incorrectly interpreted as file-level variables are more efficient, but this isn't the case as loading from a file is slower than allocating and initializing variables.

Global Variables

Is a file-level variable a global variable? In a sense it is, but there is also an important restriction. A file-level variable is only accessible to code in the same file. If you have a multi-file C project then code in other files cannot access the file-level variables in a given file.

This means that, for a single file C project, file-level variables are global variables.

What about multi-file projects?

Most non-trivial C projects use multiple `.c` and `.h` files. The compiler generally compiles each `.c` file, after combining it with any `.h` files included, in isolation to produce an intermediate `.o` file which the linker then puts together. Any file-level variables in a file are global within that file but inaccessible from other files. This is often a useful feature and can be used to provide a global variable without the risk that other files will use global variables that collide with it. If you want to share a global variable across files you need to use the `extern` keyword.

You can see that, unlike many languages, the file structure of a C project controls what is shared between the code in each of them. They are the units of isolation that allow a large project to be manageable. They are C's encapsulation in object-oriented terms. See Dive 14, Structs and Objects.

Declaring and Defining

At this point it is helpful to draw a distinction between declaring a variable, i.e. informing the compiler that the variable exists, and defining a variable, i.e. getting the compiler to allocate storage for it.

If you declare a file level variable using `extern` then the compiler allows its use in the file as a global variable, but it doesn't allocate storage for it. The linker expects to find a single definition of the variable in one of the files it is linking and it fills in the missing address in all of the files that make use of it.

The compiler converts each file or compilation unit into a possibly incomplete "object" file. Any variables that have been declared, but not defined, are included in the object file in a form that gives enough information for the linker to locate the variable in another file where it is defined. It is up to you to tell the linker which files to link together to create the complete program.

For example:

`file1.c`

```
int globalvar = 42;
int main(){…
```

As `globalvar` is declared and defined, it can be used within `file1.c`.

To use it in another file we need to use `extern` to just declare the variable:

`file2.c`

```
extern int globalvar;
functions that use globalvar
```

The `extern` informs the compiler that `globalvar` is defined in another file and that it should compile `file2.c` as if `globalvar` was an `int` but use a placeholder for its address. When the two files are linked to make the complete program, the placeholders for the address of `globalvar` are replaced by the actual address of `globalvar` as allocated in the compilation of `file1.c`.

If you plan to use global variables across multiple files then best practice is to put the `extern` declaration in a header file and make sure that this is included in all of the files that need access to the globals.

It is also worth knowing that `extern` can be used to make it clear that a function in one file is accessible from another. We tend not to use it because `extern` is the default for function declaration. If you want to make a function local to the compilation unit it is defined in then use the `static` modifier which we'll look at next.

Static

Closely related to the idea of a file-level variable, is a static variable. A static variable can be declared anywhere in a program and it has the same lifetime as a file-level variable.

That is, a static variable lives for the entire duration of the program. The difference is that a static variable has the scope appropriate to where it has been declared. That is, if you create a static variable within a function, it exists before the function executes and continues to exist after the function has exited. This sounds like a global variable, but it is only accessible from within the function. This makes a static variable look like a local variable that lives for the life of the program.

Notice that you can declare a static variable in the `main` function and this looks a lot like a global variable, but one that only the `main` program can use directly. You can also declare a static variable at file level and in this case you cannot use it as a true global variable as it is restricted to the current compilation unit, i.e. the file it is declared in.

Typically a static variable is used to store state data that persists between function calls. For example:

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

void myFunction(){
    static int myVar = 0;
    myVar++;
    printf("%d\n", myVar);
}

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

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

The only subtle point here is that the line:

```
static int myVar = 0;
```

is executed by the compiler in the sense that it reserves storage for `myVar` as part of the program file and zeros it. In this way the creation of the variable and its initialization occurs before the function is ever called and the program displays `1,2,3,4` as its result. Notice that `myVar` isn't accessible from any other part of the program.

This is slightly different from automatic variables where lifetime and scope are related. In this case the compiler enforces the restriction by refusing access from outside the function that declares the static variable.

What do you use static variables for? The simple answer is that they are ideal for storing persistent state without using file-level or true globals. The data can also be shared with other parts of the program as it is safe to return a pointer to a static. For example, we can change the previous function to:

```
int *myFunction(){
    static int myVar = 0;
    myVar++;
    return  &myVar;
}
```

Notice that it now returns a pointer to the static `myVar`.

We can use this something like:

```
int main(int argc, char **argv)
{
    int *myLocalVar;
    myLocalVar=myFunction();
    printf("%d\n", *myLocalVar);
    *myLocalVar=42;
    myLocalVar=myFunction();
    printf("%d\n", *myLocalVar);
    return 0;
}
```

This works and the static variable is changed by the `main` program.

Is this a good idea?

It is safe in the sense that the variable is allocated at a fixed location and always exists. The only downside is that the function is not re-entrant, i.e. if another thread calls it while it is being used then the static variable is not reallocated. Compare this to a function which only uses automatic local variables and allocates any other memory on the heap. Such a function is by default re-entrant.

It may be safe, but it is difficult to implement a scheme that can control the access to the static variable. One possibility is to use access functions to hide the implementation of the state representation from its clients.

It is also worth noting that there is nothing to specify how static variables are to be implemented. However, implementing them as file-level variables with restricted access is so easy, it is difficult to think of a reason why a compiler would do anything else.

Static v Heap

If you know about heap storage, see Dive 12, then you might be thinking that as static is easier to use than heap storage, why not use it in preference. The advantage of static storage is that no pointers are involved in its creation or management. Its disadvantage is that it isn't possible to manage it at all. Heap storage can be freed and resized whereas static storage is fixed.

Using static storage can also add to the size of the program file. Static storage and file-level storage is most easily implemented by being compiled into the program and how efficiently this is encoded in the source file varies, but it is possible that if you declare a 1MByte static array the size of your executable increases by 1MByte. This also increases the time it takes to load your program.

Dive 10

Pointers

With great power comes great responsibility.
Spider-Man

C has pointers and you can use them as you like. This allows for a completely free and unrestrained use of the most powerful tool in computing. It is not surprising that C programmers love pointers and bemoan their loss if cast to a less freedom-loving language.

You could say that the whole point of C is pointers.

All this is true, but using pointers well is the difference between a beginner and a master. You can also say that pointers are the biggest reason for treating C as a dangerous language. Yet languages that are proposed to replace C generally have to provide something like pointers, usually confined to "unsafe" blocks to make it practical. Even languages that are considered safe, such as Rust, have to restrict the use of pointers to a limited range of data types to make them safe. In Rust you can only create tree-like data structures in safe mode and this means that if you want to create something that isn't tree-like you have to move to unsafe code.

Pointers are considered dangerous because they give you the power to access any part of the memory, possibly including areas you don't own. They are complicated enough to make mistakes common and they are very easy to subvert.

Pointers are all about addresses and as such are at the core of low-level programming.

The Address Operator

There is a sense in which variables are already pointers. When you write:

```
int myVar = 42;
```

`myVar` has a memory address associated with it and `42` is stored in that location.

Similarly when you write:

```
myTemp = myVar;
```

`myVar` is the address of the memory location that the data is retrieved from and then stored in the memory location that `myTemp` refers to. So whether you regard `myVar` as storing `42` or the address of `42` is a matter of how you look at it. There is a sense in which a variable is a constant pointer. However, it is important to realize that `myVar` isn't a pointer and in fact doesn't even exist as an obvious entity in the compiled program – the address associated with `myVar` isn't stored anywhere. It is simply a symbolic name for an address which is used to access the data whenever you use it.

The semantics of value assignment covers up the role that the address plays in the transaction and in most languages you cannot directly access the address associated with `myVar` – but in C you can.

The address operator, `&`, will return the address associated with any variable. The idea is simple, `&myVar` returns the address associated with `myVar`, but here we hit a snag. An address is just a bit pattern like any other item of data, but the format varies between different machines. The number of bits used for an address changes and so do addressing schemes. Some machines have a linear address space running from *`addressmin`* to *`addressmax`*, others use multi-level addressing with page or segment number followed by offset from the start, and so on. When programming in C, however, you don't have to delve into the details. You can simply treat `&myVar` as a simple address.

If you try something like:

```
int main(int argc, char **argv)
{
    int myVar;
    int myAddress;
    myAddress=&myVar;
    printf("%d\n",myAddress);
    return 0;
}
```

you will see a value displayed which is the address of `myVar`, but you will also probably see a warning that amounts to assigning an address to an `int` isn't a good idea. An automatic cast from an address to an `int` saves you from the problem, but in most cases it doesn't really help. You take the address of something because you want to access it, but when the address is stored in an `int`, all you can do with it is integer arithmetic.

The Pointer

We need a new data type.

The clever part is that C doesn't introduce a single new data type "the address", but a modifier, * the dereference operator, that can be applied to any data type to produce an address that references that type. This is called indirection and the resulting data type is a "pointer" or a "pointer to the type". For example, `int myVar` is an `int` whereas `int *myPointer` is a pointer to an `int` and so on. Notice that this is more subtle than simply defining a type that holds an address.

Both `float *myFloatPointer` and `int *myIntPointer` are going to be the same number of bits and essentially the same, but the compiler remembers that the first is an address of a `float`, typically 8 bytes, and the second is the address of an `int`, typically 4 bytes.

That is, the pointer records not just the fact that it is an address, but also the nominal size and configuration of the entity it is pointing at. It is important to understand that this is only true at compile time. The compiler uses the information to work out how to treat the data that a pointer references. At runtime we only have the address of the entity.

Notice that the * binds to the variable name not the type. That is, the declaration is:

```
int (*myIntPointer)
```

because of this you can write things like:

```
int *myIntPointer, myInt, *myIntPointer2
```

and so on. Compare this to the way the type is written in cast:

```
(int*) myPointer
```

You can still write it as:

```
(int  *) myPointer
```

and imagine that the * still binds to the variable.

Dereferencing

Now that we have a pointer type, we can make use of addresses derived using the address operator, for example:

```
int myVar = 42;
int *myIntPointer;
myIntPointer = &myVar;
```

Now we have the address of `myVar` stored in `myIntPointer`.

If you write:

```
temp = myIntPointer;
```

then the address is copied into `temp`. How then to retrieve the value that is stored at the address? The answer is the dereference operator `*` which we have already met in the declaration of a pointer. The dereference operator is the inverse of the address operator in that it accesses the value that the address references. For example:

```
int myVar = 42;
int temp = *&myVar;
```

stores `42` in `temp` as `&` gives the address of `myVar` and `*` then gives the value stored at that address. Of course this isn't actually useful. You can use the dereference operator on the left of an assignment:

```
int myVar = 42;
int *myIntPointer;
myIntPointer = &myVar;
*myIntPointer = 43;
```

This stores `43` at the same location as referenced by `myVar`. We now have reference semantics in C – almost.

Lifetime Problems

The existence of pointers in C brings with it a new problem – "use after free". This refers to the idea that now we can reference storage in a way that isn't controlled by the system. When you use an automatic or static variable, its lifetime is such that if you try to access it within its scope then it exists. Once you have pointers you can make the mistake of using a reference after the variable no longer exists – after it has been "freed". This problem gets much worse once we introduce the use of the heap – see Dive 11 - but it is still a problem in this simpler situation.

For example:

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

int *myFunction(){
    int myVar=42;
    return &myVar;
}

int main(int argc, char **argv)
{
    int *myPointer=myFunction();

    printf("%d\n",*myPointer);
    printf("%d\n",*myPointer);
    return 0;
}
```

In this case the function returns a pointer to an auto variable which is, of course, created on the stack and then destroyed as soon as the function returns. The pointer that the `main` program receives has the address of the memory location on the stack that the data once occupied. The result is use-after-free.

This is exactly the problem discussed in Dive 8 where the blame was placed on the use of local variables – in fact the culprit is much more the use of a pointer to return a reference to something that possibly no longer exists. As explained in Dive 8 the worst part of the error is that the error often goes unnoticed because the data stored on the stack isn't overwritten for a while and we seem to get the correct result. For example in this case the first `printf` is very likely to display the expected value of `42` but in the process of displaying the value it is very likely to overwrite it, causing the second `printf` to display a spurious value.

Notice the problem goes away if `myVar` is declared as `static`, see Dive 9, as the variable exists even when the function has returned.

The C programmer is free to do almost anything with a pointer and this is the power that demands the responsibility.

In fact, you can even create a "use before allocate" error. For example:

```
int *myPointer = 4796436;
printf("%d\n",*myPointer);
```

In this case we are attempting to access an arbitrary memory location by loading an `int` address into the pointer. This is unlikely to work as most modern hardware has protection against a program accessing a memory area it doesn't own, but if you pick an address that is within the program's address space it does work.

This sort of code is used a lot in hardware applications where the program needs to access registers at fixed memory locations. The point is what appears at first to be something so silly that no programmer would ever or should ever do it, has a context in which it is perfectly reasonable and sane. A compiler writer who thinks that this is so serious an error as to disallow it would make many a C programmer unhappy.

Operator or Type?

If you are anything like me you will be slightly perturbed that the dereference operator is also a type indicator. The two expressions `int *myVar;` and `int temp = *myVar;` seem like two different things but not if you look at the declaration in the right way and treat the `*` as the dereference operator.

When you write:

```
int myVar;
```

this declares `myVar` to be an `int`. In the same way, when you write:

```
int *myVar;
```

this declares `*myVar` to be an `int`. So, if `*myVar` is an `int`, `myVar` (without the operator), has to be a pointer to an `int`. This is neat, but it doesn't quite work. When you write the cast `(int*)`, which means cast to a pointer to `int` as now the `int*` seems to operate as a unit which says "pointer to int".

But as already pointed out, you can always regard `(int *)myVar` in the same way as the declaration i.e. if `myVar` is cast to an `int*` then `*myVar` has to be an int. It's just not as neat.

Pointers, Arrays and Heaps

It is reasonably true that pointers only come into their own when used in conjunction with the heap. However, we have already seen that pointers can be useful with a static variable and before we encounter the heap we need to understand how pointers give rise to arrays.

Dive 11

The Array and Pointer Arithmetic

Real programmers count from zero.

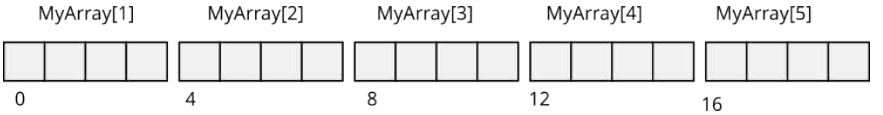
Anonymous programmer

The array is the first data structure every programmer meets. It is essential and basic and is the prototype for every other data structure that attempts to usurp it. The array is basic because it's just a section of memory, almost without any additional structure.

The Storage Mapping Function

Most C programmers know about arrays, but it is worth recalling what lies behind. An array is a set of identical data types accessed by an index. In C the only type of index allowed is an integer and the storage is contiguous – i.e. with no gaps. This is because of the way that an array is implemented using a storage mapping function, SMF.

Suppose you have an array of five integers and each is four bytes. Assume that the ints of the array are laid out in a contiguous block of memory:

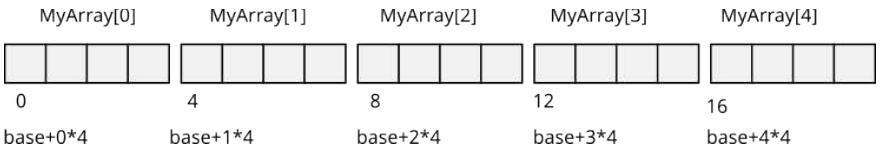


If each `int` takes four bytes, then the first `int` is stored at `base`, the address of the start of the array. The second is stored at `base+4`, the third is stored at `base+8` or `base+2*4`. You can see that in general the ith element is stored at:

```
address = base + (i-1)*4
```

You can get a slight simplification by starting to count the elements from `0`:

```
address = base + i*4
```



Now element `0` is at base, element `1` at `base+4` and so on.
And, yes, this is the reason that programmers count from zero not one.

Going beyond the simple `int` array, you can see that, in general, if the repeated element in the array takes `B` bytes to store then the SMF for the array is:

```
address = base + i*B
```

The same idea works for multi-dimensional arrays.

For example, a two-dimensional array with n rows and m columns can be implemented using the SMF:

```
address = base + i *m *B +j*B
```

as `m*B` is the size of a single row – this is often called the "stride" because it's the amount you have to jump to move to the same element in the next row.

That's it, that's all there is to arrays. Things really only get more complicated when you need an array that is bigger than the available memory or when there is some special requirement to be able to reorganize the array efficiently. Large arrays are generally broken down into pages, each of which will fit into memory. In this case the SMF is factored into an expression that gives the page number and the offset within the page. Arrays that need to be reorganized are generally implemented as linked lists, another major use case for pointers.

In C, an array is a contiguous block of storage that fits into memory i.e. does not need to use secondary storage.

Index Notation

All you need to implement an array is a block of memory and a pointer to the start. You can then use a storage mapping function to access any particular element. This is easy, but most computer languages, C included, provide some syntactic sugar to make working with arrays possible, even before you meet pointers.

An array is declared using:

```
type name[number];
```

This allocates enough memory to store *`number`* copies of the specified type, for example:

```
int myArray[10];
```

In this case the compiler allocates 40Bytes, sufficient to store 10 4Byte ints.

Once you have an array allocated you can access any element using indexing. To access the ith element you write:

```
myArray[i]
```

which is, of course entirely equivalent to accessing the `int` starting at

```
base+i*4
```

Notice that as we are using the simpler SMF, the first element is:

```
myArray[0]
```

and the last is:

```
myArray[9]
```

This mismatch between the number of elements and the index of the last element is something that really bothers beginners who often, incorrectly, think that:

```
int myArray[10];
```

means that the last element is `myArray[10]`. It is also the reason that `for` loops that process the entire array are written to stop at *`number-1`*, rather than *`number`*. To avoid both of these confusions some languages, Basic for example, work with arrays that use `1` as the start rather than `0`.

Arrays Are Pointers

In other languages the distinction between an array and something you might construct using an SMF is deeply hidden in an effort to make the language as high-level as possible, but C isn't a high-level language. In C we make a point of an array being a pointer.

In C the instruction:

```
int temp = myArray[0];
```

is equivalent to:

```
int temp = *myArray;
```

The variable that represents the array is a pointer and can be used with either indexing or dereferencing. In fact, any pointer can be used with indexing and, in this sense, not only is every array a pointer, but every pointer is an array. For example:

```
int  *myPointer = myArray;
myPointer[1] = 42;
```

`myPointer` isn't declared as an array and yet you can use array index syntax to access an element as if it was an array.

Notice that arrays aren't completely like pointers. Unlike a pointer, the array name isn't a variable and as such cannot be modified. For example;

```
myArray = myPointer;
```

is illegal. In this respect, arrays are more like constant pointers.

Another important difference between an array and a pointer is the way `sizeof` works. In the case of an array `sizeof` returns the number of elements in the array, whereas in the case of a pointer `sizeof` returns the size of the pointer, typically four bytes, and not the number of elements that it points at. This is a difference imposed by the compiler in that it "knows" the size of

the array at compile time and adjusts `sizeof` accordingly. If you pass an array into a function then the array is treated as a pointer - it is said to "decay" to a pointer, but really all that happens is that the compiler loses track of the fact that it is an array and `sizeof` returns the size of a pointer. These idiosyncrasies are imposed as an attempt to make arrays behave more like you would expect them to.

Some programmers are very firm in their opinion that an array is not a pointer and, as long as you can see that point of view, you are safe to understand that pointers and the SMF are the basis of the C array.

Pointer Arithmetic

The fact that pointers are arrays and arrays pointers leads us to implement pointer arithmetic in a very particular way.

Let's ask, what is the pointer equivalent of:

```
myPointer[5] = 42;
```

If `myPointer` references the start of the array then, using the SMF idea, we can see that `myPointer`[5] is the same as:

```
myPointer + 5*sizeof(int)
```

To make this simpler pointer arithmetic is defined as:

```
myPointer + 5 = myPointer + 5*sizeof(int)
```

where the compiler takes care of multiplying by the size of the element type. That is, in general:

```
myPointer + i = myPointer + i* sizeof(*myPointer)
```

When you add or subtract from a pointer the arithmetic is done in units of the size of the type that is referenced.

Notice that you can apply pointer arithmetic to an array name. For example:

```
myArray + 5;
```

is exactly the same as:

```
myArray[5];
```

but recall that `myArray` behaves more like a constant pointer and you cannot assign to it.

Pointer to Array

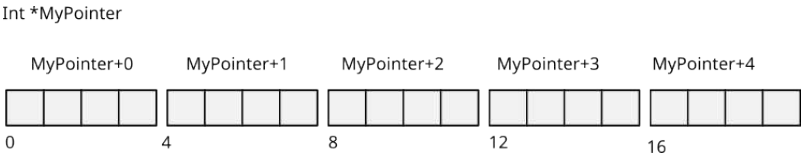
There is a subtle difference in C between a pointer to a type and a pointer to an array of the type. In the first case the pointer references the type and pointer arithmetic moves the reference on by the size of the type. This makes the pointer suitable for the implementation of an array of the type.

Compare this to the idea of a pointer to an array of the type. This was introduced in C89. It is the array of the type that is the fundamental element that the pointer references. If you now add one to the pointer, then it moves on, not by the size of the type, but the size of the array – as the array is the element.

If you write:

```
int *myPointer;
```

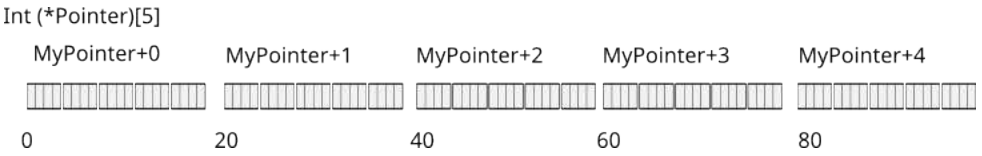
then `myPointer` references an integer and adding `1` to it moves it on by the `sizeof int`. This makes it a way of implementing an array of `int`.



Compare this to:

```
int (*myPointer)[5];
```

which declares a pointer to an array, each element of which is an array of five ints.



It is a strange notation and probably would be better if it was written:

```
int[5] *myPointer:
```

but it isn't. The type "pointer to array of five ints" is written `int (*) [5]` which is the form you would use in a cast.

The difference is that now when you add `1` to `myPointer` it moves on by five times the size of `int`, i.e. `20`. Notice that this is the array equivalent of creating a pointer to a struct and it clearly has something to do with multidimensional arrays, see later and Dive 11.

Casting Pointers

Pointer arithmetic is all about implementing the storage mapping function for the array using pointers and, in most cases, this is exactly what you want. Just occasionally, however, you need to access parts of a data type rather than the whole thing and to do this you need a pointer that points "inside" the data, not just at the start.

For example, suppose you want to access the individual bytes that make up an int. Getting a pointer to the int is easy, but as it is an int pointer adding one to it moves it on by four bytes. If you only want to move it on by a single byte you need to use a cast:

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

int main(int argc, char **argv)
{
    int myVar = 0x01020304;
    char *myPointer = (char *)&myVar;
    printf("%d\n", *(myPointer+1));
    return 0;
}
```

By casting the address to a `char` pointer adding `1` really does only add `1` . You can do this on a temporary basis by casting the pointer:

```
int *myPointer=&myVar;
printf("%d\n", *((char *)myPointer+1));
```

Of course, what you actually see displayed depends on the byte order – big or little endian. For more complex data structures, the exact layout in memory makes a difference.

More Than One Dimension

If you know about arrays then you will also know that they come in different shapes. The sort of array that we have been looking at is usually called "one-dimensional" or a "vector" because it has only a single index. You can have an array that has more than one index. A two-dimensional array has two and it is generally thought of as a table of rows and columns or as a matrix. For three dimensions we have a cube and beyond this things become difficult, if not impossible, to imagine.

Higher-dimensional arrays are often called "tensors", but this name comes from physics and math. As far as programming is concerned, an n-dimensional array is simply an array which accepts n indices to identify an element.

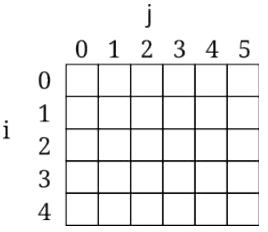
Some languages have special syntax for defining multi-dimensional arrays, but C just reapplies the basic array principle as many times as required. The

point is that a two-dimensional array is a one-dimensional array where each element is a one-dimensional array. If you want to think in terms of the rows and columns of a matrix, then the C array is a one-dimensional array of rows. This is generally called row-major order as opposed to column-major order used by Fortran, Octave and so on.

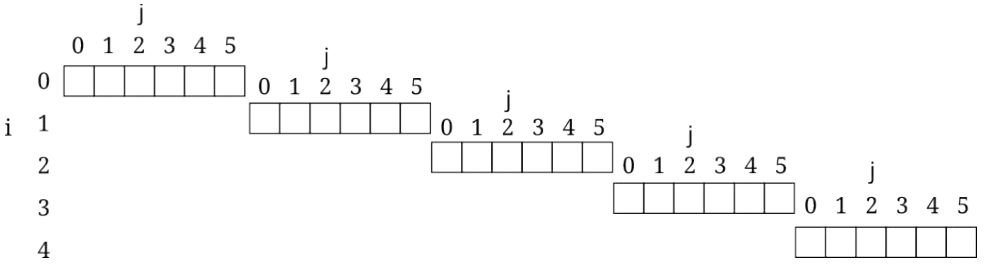
The syntax to create a two-dimensional array follows from this description:

```
int myArray[5][6];
```

which creates an array with 5 rows, each one consisting of an array of 6 ints – the columns.



You can think of this as creating a one-dimensional array where each element is in itself an array of 6 ints.



`int myArray[5][6];` can be thought of as `int[6] myArray [5]`, i.e. an array of five elements each of which is an `int[6]`. Of course, this isn't valid C - it is just intended to give you some idea of why the syntax is as it is.

Accessing an element of an array also follows the "double index" pattern:

```
myArray[i][j]
```

refers to the element in the ith row and the jth column. Again it takes the form of the jth element of the array that is the ith element.

This all works with more dimensions. For example, a three-dimensional array is declared as:

```
int myArray[5][6][8];
```

and is composed of an array of 5 arrays of 6 elements each of which is an array of 8 ints. To access an element you would use:

```
myArray[i][j][k]
```

the ith "plane", jth row and kth column.

Beyond three dimensions, it is better to just stick to the index notation and stop worrying about how to imagine the arrangement in space.

Pointers to Pointers

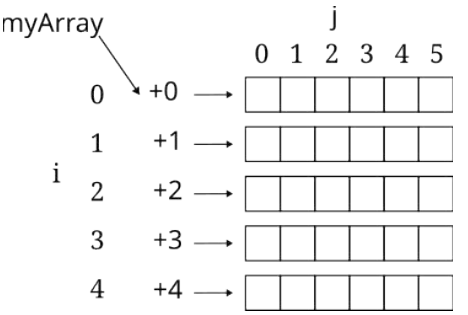
Arrays and pointers are almost one and the same, so what happens in the case of a multi-dimensional array? What is the pointer equivalent of:

```
myArray[i][j];
```

The answer is slightly more complicated than you might expect. After:

```
int myArray[5][6];
```

`myArray` is a pointer, but in this case it is a pointer to a pointer to an array of 6 `ints`:



So now it should be clear that:

```
myArray
```

is a pointer to an array of 6 ints and so:

```
myArray+1
```

is a pointer to the second row of the array as pointer arithmetic moves it on by 6 ints, the size of a single element, and in general:

```
myArray+i
```

is a pointer to the ith row of the array.

As `myArray` is a pointer to any array of 6 ints, dereferencing it gives the array of 6 ints, which is, of course, a pointer to `int`. So:

```
*(myArray+1)
```

is a pointer to `int`, i.e. it is a one-dimensional array.

Thus:

```
*(*(myArray+i)+j)
```

is the jth element in the ith row.

Notice that all this works because the type of `myArray` is `int (*)[6]` as this specifies how much to add in pointer arithmetic to move on to the next row. Also, even though this is a pointer to a pointer, there is no other pointer actually involved. There is only `myArray` pointing to the start of the entire array. The trick is all based on changing the type of the pointer to generate different pointer arithmetic so as to implement the storage mapping function needed for the dimension of the array. That is, `myArray` is of type `int (*)[6]`, so `myArray+i` implements `myArray+i*6*sizeof(int)` and `*myArray` is of type `int [6]` and so `(*myArray)+j` implements `myArray+j*sizeof(int)`, which, when you put these together as `*(*(myArray+i)+j)`, implements:

```
myArray +i*6*sizeof(int) +j*sizeof(int)
```

which is the SMF for the two-dimensional array.

This idea works no matter how many dimensions that array has. If the array is three-dimensional:

```
int myArray[2][3][4]
```

then `myArray` is of type `int(*)[3][4]`, i.e. an array of three elements each of which is an array of four ints. When you add one to `myArray` pointer arithmetic moves it on by three by four ints. Next we have `*myArray` which is of type `int(*)[4]`, i.e. an array of four ints. Adding one to this moves the pointer on by four ints and finally `**myArray` is simply a pointer to an `int` array. Thus to access the element given by `i`, `j`, `k` we use:

```
*(*(*(myArray+i)+j)+k)
```

It is difficult to follow the details, but it works as advertised.

Finally, notice that you can get a simple pointer to the start of the entire array using:

```
int *myPointer=(int *) myArray;
```

Using this you can implement your own SMF or access the array as a single dimension. It is a general principle that you get a pointer to the start of the array whenever you convert the array to a pointer.

You can also use casts to alter the arrangements of rows and columns. For example, if you start out with a five by six matrix then you can convert it into a three by ten matrix using:

```
int myArray[5][6]
int (*myPointer)[10];
myPointer = (int (*)[10]) myArray;
printf("%d\n", myPointer[1][2]);
```

Once you realize that the whole multi-dimensional array trick is about using casting and pointer arithmetic to implement SMFs, then suddenly everything seems so much more obvious.

Strings As NULL-Terminated Arrays

Strings are the lifeblood of the user interface and C is often criticized for doing them very badly indeed. It has to be admitted that strings are not C's best feature, but they are efficient, close to the machine and fast.

The reason they are as they are is simply that when C was being developed memory was in short supply and text demands a lot of memory. Modern languages use more sophisticated ways of representing strings – storing additional information along with each string such as how many characters it contains and adopting immutable strings. None of these approaches can compare with the simplicity of the C string.

A C string is simply an array with a sentinel value marking its end. The sentinel is always `NULL` i.e. zero, because zero doesn't correspond to any displayable character. This is one occasion when `NULL` can be safely used as an end marker.

`NULL` may be safe, but not having a `NULL` is definitely not safe. The main danger in using C strings is that you will encounter a string that is not `NULL`-terminated and as a result you will read or write beyond the end into memory you don't own. Of course, by good design and programming you can avoid this sort of problem, but when you are processing strings obtained as input, you can be the subject of an attack that you have to defend against.

A string is a `NULL`-terminated array and as such nothing special. However, C does provide some additional features to make working with strings easier. The most obvious is the way string literals can be used to initialize `NULL`-terminated `char` arrays. For example:

```
char myString[50] = "Hello World";
```

stores the characters in `myString` and ends with a final `NULL`. Notice that, in this case, the array is longer than the number of characters to be stored.

You can count the number of characters in the literal and simply set the array to the correct size or you could let the compiler do it for you:

```
char myString[] = "Hello World";
```

As `myString` has a final `NULL` element it might be one bigger than you expect based on a simple character count.

What makes `NULL`-terminated strings so nice to work with is that you can scan them using a `for` loop. For example, you can find out how many characters are in a string using:

```
for(int i=0; myString[i]; i++) count++;
```

Notice the use of `myString[i]` as the stopping condition. Another good reason for using `NULL` as the sentinel, is that it is also interpretable as false.

This gives you some idea of why `NULL`-terminated strings seemed such a good idea back in the early days of C. It also indicates why they aren't quite so good – you have to scan the entire string to find its length and what happens if there isn't a `NULL`.

You may not be careless enough to write loops that might not end, but there is still the problem of using such a thing within a library function. For example, the standard function `strlen` returns the length of a string and seems innocent enough, but it uses the same sort of `for` loop to scan the string and to count characters. If there is no `NULL` then the loop doesn't end and you overrun the end of the array. Notice that the sizeof operator will tell you the length of the array that holds the string i.e. `sizeof(`*`string`*`)`. This returns at once and doesn't scan the string so it is 100% safe.

Nearly all of the standard string handling functions have this defect. There is a much safer version of each of these functions indicated by an "n" in the middle of the name. For example:

```
strnlen(str, size);
```

will return the length of `str` as long as it is less than `size`. That is, the scan stops after `size` characters have been examined and no `NULL` found. The only problem is that this is not a standard C function. It's a Posix standard, i.e. part of Linux/Unix.

Similarly the function:

```
strcpy(dest,src);
```

will copy the string in `src` into `dest` including the final `NULL`. A safe version is:

```
strncpy(dest,src,size);
```

which stops after `size` characters if it hasn't encountered a `NULL` in `src` before this. Its behavior is slightly more dangerous than you might think in that if `src` doesn't have a `NULL` in its first `size` bytes then `dest` will not be

`NULL`-terminated. In other words, `strncpy` limits the size of the scan, but doesn't promise that the result is `NULL`-terminated. To be on the safe side, you can protect yourself by always storing a `NULL` in `dest[size-1]`.

In practice, you can protect against C strings that lack a terminating NULL by always limiting operations to the size of the arrays used to hold the strings. It is common to make the arrays one byte larger than they need to be and initializing them with a final `NULL` to act as a sentinel of last resort.

C strings may be unsafe without due care and many adopt an alternative representation by creating a struct with a `string` and a `len` field. You can generally use the safe string functions with these by splitting out the parts of the struct. If you plan to do this it is worth finding and using an improved C string library, such as the SDS library or the ICU4C library with Unicode support.

Dive 12

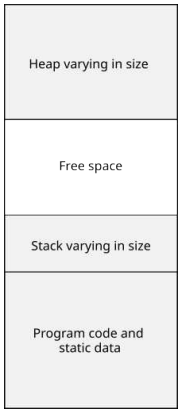
Heap, The Third Memory Allocation

Facts are a heap of bricks and timber. It is only a successful theory that can convert the heap into a stately mansion
Isaac Asimov

So far we have seen two different types of memory allocation, file-level and automatic. Now it is time to meet the heap, the most sophisticated and direct form of memory allocation available to the C programmer. In many ways advanced C programming only really starts when you use the heap and yet the heap solves so many problems, it makes things simpler. To make good use of the heap you have to have a good grasp of pointers and how to use them.

The Heap

Usually, but not always, a computer's memory is divided up such that the program and all of the file-level data are stored in the lowest portion. Then comes the stack which is allocated as needed and grows and shrinks as auto variables are created and destroyed. The heap is the remaining available memory usually organized as a single contiguous block starting at the top of available memory that can be allocated for use in blocks of a specified size.



The growing and shrinking of the stack is extremely efficient in the sense that only the exact amount of memory needed at any given time is used and nothing special has to be done to release previously used stack memory.

This isn't the case for the heap, which quickly becomes a patchwork of allocated memory and released memory. This doesn't matter as long as there is a free block of heap memory large enough to satisfy the current request for memory. To keep large areas of free memory available the heap has to be subject to some sort of garbage collection. This is the responsibility of the operating system and can be mostly ignored by the C programmer.

In principle, if the heap has the amount of memory you have requested free, then it should be available as a contiguous block. However, there can be consequences of the operating system managing the heap such that occasionally it will need to make use of the CPU to rearrange the heap to coalesce separate free blocks which are too small on their own and this could introduce a slowdown in your program. This is less of a problem in a system with multiple CPUs.

Notice that another big difference with the heap is that the programmer is responsible for allocating and deallocating memory. This is usually explained to be the most difficult part of using the heap. In fact allocating memory is generally easy and deallocating it is slightly trickier in complex situations where different parts of the program make use of the same block of memory. It can be difficult to be sure that an allocated block isn't still in use by some other part of the program when you decide to deallocate it.

Another big difference between file, auto and heap variables is that file and auto variables are allocated automatically – you simply declare them and they are created for you using the amount of memory necessary for the type. A heap variable on the other hand requires you to allocate a block of memory large enough to store its type and then associate that type with it so that can be easily used.

All of this depends on the use of pointers and sometimes some quite sophisticated or clever use of pointers.

Malloc and friends

There is a set of C functions concerned with working with the heap. These make use of facilities provided by the operating system to manage the heap. The best known is `malloc`, standing for memory allocation. This simply returns a pointer to a block of memory of the required size:

```
void *malloc(size_t size);
```

`size` is in bytes and if it works it returns a pointer to `void` to the start of the block of memory allocated. It is important to know that the block of memory is not initialized in any way and this is the basis of many exploits as you can use it to discover data that a previous program stored in the memory.

If `malloc` allocates a block of memory `free` releases it back to the heap. There are some interesting details of free that if better understood could save us from memory errors. If you call `free` on a pointer to memory that has already been freed then the result is undefined behavior. However if you call free with a `NULL` pointer then no operation is performed. `NULL` is simply a stand-in for zero but most systems regard an address of zero as not being an address you can allocate using `malloc`. So if you always set a pointer to `NULL` after freeing the memory you can use this as a flag to indicate that the memory has been freed meaning that even if another part of the program tries to free it a second time there will be no problem – as long as it uses the same pointer.

There is no doubt that you only need `malloc` and `free` to work with the heap, but there are three common useful extras that can make life easier.

A common requirement is to create a block of memory large enough to store an array of identical elements. This is usually achieved using:

```
p = malloc(sizeof(type)*N);
```

where `N` is the number of elements, but it can be written:

```
p = calloc(N, sizeof(type));
```

The saving is tiny, but there is another, almost more important, difference between `calloc` and `malloc` – `calloc` zeros the block of memory returned and `malloc` doesn't. If you need a zeroed block of memory then it might be worth using:

```
p = calloc(1,sizeof(type));
```

If you need to change the size of a block of heap memory, you can use:

```
p = realloc(ptr, size);
```

This changes the size of the block referenced by `ptr` to the size specified. The existing data isn't changed and any new memory locations are not initialized. The block of memory can be increased or decreased in size. If `ptr` is `NULL` then a new block is allocated. If `ptr` is not `NULL` and `size` is `0`, the block is released. If the block has to be moved to make the resize possible the original block that `ptr` referenced is freed and `ptr` is set to `NULL`. In practice always use `p` rather than `ptr` after `realloc`.

There is also:

```
p = reallocarray(ptr,N,sizeof(type));
```

which works like `realloc`, but the size is specified in terms of the size of the element and number of elements. Apart from this it works in exactly the same way and in particular any additional memory is not initialized.

Casting

Apart from the tricky problem of freeing memory in an organized fashion and making sure that you don't try to use the memory after freeing it, using the heap is easy.

But how do you turn the block of memory into something useful?

All we have is a block of memory, a block of bits, but the repeating message of using C is that everything is just a bit pattern. What tells the compiler how to treat the bit pattern is its type. So all we have to do is cast the pointer to the type we want to use. For example, if you want to allocate an `int` on the heap you would use:

```
int *p = (int*) malloc(sizeof(int));
```

Now `p` points at the start of a block of memory, usually 4 bytes, and the system will treat this as an `int`. That is you can write things like:

```
*p = 42;
int sum = *p + 42:
```

In fact, as long as you remember to dereference it, `p` behaves exactly like a standard `int` – you could almost tell a beginner that `*p` was the variable name! There are of course differences and they are important.

The first is that the lifetime of the variable, the block of heap memory, is under your control. It exists from the moment the `malloc` is executed until you explicitly call `free` to release the block. Notice that the lifetime of the pointer that references the block follows the usual rules, which aren't the same as for the lifetime of the block. If it is a file-level variable then it lives for the entire program. If it is an auto variable then it lives for the same time as the function it is declared in. What this means is that a block of heap memory can become orphaned with no pointers referencing it.

Consider the following function:

```
void leak(){
  int *p= (int*) malloc(sizeof(int));
};
```

Calling this function allocates a small block of heap memory, but when it terminates the pointer is destroyed and the block has no references to it. This is a memory leak as the block can never be freed as the program has no references to it. Many other languages implement garbage collection algorithms that look for blocks of memory that have been orphaned to automatically free them. This is a very time-consuming thing to do and not foolproof.

The C language prefers to let you manage the freeing of heap memory – you are C's garbage collector.

If the lifetime of the pointer corresponds to the lifetime of the heap block then there isn't much point in using it as this is true of an auto variable. In practice it is common for functions that create a heap variable to pass the pointer back to the calling program for it to use and thus the lifetime of the heap variable is longer than that of the function.

Dynamic Allocation

One of the big advantages of using the heap is that you can ask for as much memory as you need at the time that you need it. Auto variables allocate memory as the program runs, but the size of the allocation is fixed at compile time – there are exceptions, see later.

For example, when you declare an array the compiler has to know how big the array is. That is, in C89:

```
int myArray[10];
```

is legal:

```
int myArray[N];
```

isn't. C99 relaxes this so that N can be determined at runtime. This is a so-called "variable length array", VLA, and compilers such as GCC will let you use a VLA, even if you ask to use C98. The VLA idea is so useful that it has spread beyond its original bounds, even if C11 marks it as optional. This said, it is worth noting that the Microsoft C compiler doesn't support it and if you want to write portable code it is a good idea to avoid it. For this reason the Linux community spends a great deal of time removing VLAs from its code base.

So how can you create a dynamic array without VLAs? The answer is you can use `malloc`:

```
int N = 10;
int *myArray = (int*) malloc(sizeof(int)*N);
myArray[0] = 42;
*(myArray+1) = 43;
printf("%d\n", myArray[1]);
```

This allocates a one-dimensional array of 10 `int`s which can be used with indexing or with pointer arithmetic. As always, the block of memory remains allocated until you free it and the lifetime of the pointer `myArray` is determined as usual.

Dynamic Multidimensional Arrays in C89

What about a multidimensional dynamically allocated array? Here the problem is that before C99 there was no way to specify the size of the dimensions and these are needed to compute the SMF. In this case the only simple solution is to implement the SMF directly. As we will shortly discover, in C99 and later there is a better solution.

For example, for an `M` rows by `N` columns array:

```
int *myArray = (int *)malloc(sizeof(int) * N * M);
int *p = myArray + (i * N + j);
printf("%d\n", *p);
```

This displays `myArray[i][j]`, but notice that indexing doesn't work in this case as the compiler has no idea of the "shape" of the array. You can't use the array pointer cast that works with a fixed size array because until C99 there was no way to specify a pointer to an array of a given size – see later and see Dive 11.

The best you can do is to create a pointer to an array with all but one dimension fixed in size. The reason that this is so is that for a 2D array the compiler needs to know the size of each row, i.e. `N`, so that it can compute the SMF, but once we have selected the row the pointer arithmetic is in terms of `sizeof(int)` and you don't need to know the number of rows to work out the SMF. You can see that this is true just by looking at the SMF for a 2D array:

```
address = base + i *N *sizeof(int) +j*sizeof(int)
```

where `N*sizeof(int)` is the size of a single row. The number of rows, `M`, isn't referenced, which means you can implement the SMF without knowing it.

For example, if the array is `myArray[M,5]` then `myArray` is a pointer to a one dimensional array of size `5` and `*myArray` is a pointer to `int`. Clearly we cannot do pointer arithmetic to work out the SMF but `M` isn't used in the calculation. What this means is that we don't have to specify the first dimension in the dynamic array:

```
int M = 6;
int (*myArray)[5] = (int (*)[5]) malloc(sizeof(int) * 5 * M);
myArray[1][1] = 42;
```

Notice that only the value of `N` is fixed at 5 in the declaration of the array and the type used is a pointer to arrays of size 5 (see Dive 11 for more on pointer to arrays). This has to be so to allow `myArray[i][j]` to be implemented as:

```
*(*(myArray+i)+j)
```

with my `Array+i` adding `i*5*sizeof(int)` to the pointer. This code allows you do dynamically allocate an array equivalent to:

```
int myArray[M][5]
```

Notice that while you don't have to fix the size of `M`, you do have to check that you don't try to access `myArray[i][j]` for `i>=M`. If you do you will run off the end of the array and the allocated memory.

Reallocating

If it turns out that the array that you are using should have been bigger, you can simply reallocate the block of memory to be bigger. The same technique works if you need to make the block smaller, but this is a less common action. The principle is to use:

```
p = realloc(ptr, size);
```

where `ptr` is to the existing block and `p` is the pointer to the resized block.

The operating system will first see if there is enough free memory at the end of the block and if there is it is simply added to the existing block. If there isn't, then the operating system searches for a free block big enough and copies the data from the old to the new block. The operating system may also attempt to defragment the heap to find a block large enough.

The reallocation idea is simple as long as you are working with one-dimensional arrays. In this case the additional block of memory simply adds to the length of the array. If you are working with multi-dimensional arrays things are more complicated. Adding memory could well change the SMF you need to use and as a result elements that were in their correct place might well be in the wrong place, even though the reallocation hasn't moved them.

The one reallocation that is "safe" is adding rows because that dimension isn't used in the SMF. For example, to reallocate a `[M][N]` array to `[M+1][1]` all you have to do is:

```
int *myArray = malloc(sizeof(int) * M * N);
myArray[1][0] = 42;
myArray = realloc(myArray,sizeof(int)* (M+1) * N);
printf("%d\n", myArray[1][0]);
```

It just works - you see `42` displayed, and you have an additional row to work with. However, if you try:

```
myArray=realloc(myArray,sizeof(int)*M*(N+1));
```

then you still see `42` displayed, but you cannot correctly access the new column as the SMF is still working with `M` columns. To reform the array to its new structure, you need a different pointer type, assuming `N` is `5`:

```
int (*myArray2)[6])=realloc(myArray,sizeof(int)*M*(N+1));
printf("%d\n", myArray2[1][0]);
```

Now we have an array with `6` columns and you will not see `42` displayed because this is now stored in `myArray2[0][5]`.

Reforming arrays is easier when you make use of VLAs, see below, but you always have to move elements if you add additional columns.

The Variable Length Array

The Variable Length Array, VLA was introduced in C99 and it is optional in C11 but mandatory in C23. Basically it is an automatic implementation of the dynamic array allocation using `malloc`, but you simply declare the array in the usual way and the compiler does the rest. For example, the dynamic array created in the previous section can be declared as:

```
int myArray(M,N);
```

The resulting array behaves just like a standard array and you can use index notation or pointer arithmetic to access it.

There are some important differences, however. A VLA can only be declared as an `auto` and its lifetime is determined by this. When the variable goes out of scope, the VLA is destroyed. You don't have to worry about using `free` and this is one of the claimed advantages of a VLA, but it also restricts its usefulness. Obviously you cannot initialize a VLA at compile time and it is re-created, optionally with a different size, each time the block of code that contains it is re-entered. If you use `goto` or `switch` to jump into the scope of a VLA then it might not be allocated and if you use `longjmp`, see Dive 17, – then data can be lost.

VLA Type and Dynamic Multidimensional Arrays

As well as the VLA allowing variable dimensions, the pointer to an array type has been extended to allow runtime expressions. That is, in C99 and later, you can write things like:

```
int (*myArray)[M]
```

where `M` is an expression that is evaluated at runtime. Previously we had to specify a constant for `M` to allow the compiler to work out what to add in pointer arithmetic. In this case, `myArray+j` is equivalent to `myArray+j*M*sizeof(int)`.

What this means is that we can now extend our dynamic array allocation using `malloc` to include multi-dimensional arrays that are defined at runtime. For example:

```
int M = 6;
int N = 5;
int (*myArray)[N] = (int (*)[N]) malloc(sizeof(int) * N * M);
myArray[1][1] = 42;
```

Notice this is the same as the previous example but now the value of `N` is used, making the type of `(*myArray)[N]` only defined at runtime. Even though we are using a variable array type, this is not a VLA – it isn't automatically deallocated and we have to remember to free the memory.

Heap or Stack

You may have noticed that VLAs have the same lifetime as an `auto` variable. This means that they are a natural for stack storage rather than the heap. Indeed, the majority of VLA implementations use the stack rather than the heap to store VLAs.

You can even allocate memory on the stack manually using:

```
void *p = alloca(size);
```

which returns a `void` pointer to a block of memory on the stack. This function is part of the standard C library, but not part of a standard. It is compiler- and machine-dependent and these are reasons enough to avoid it. As with VLAs, it can only be used with an `auto` variable, but unlike VLAs the block of memory is released only when the function that created it terminates, not when the pointer variable goes out of scope. You should never attempt to release stack memory using `free()`.

Which is better: stack or heap?

This is a question that is so machine-dependent it is impossible to give a single answer. Stack memory is often in short supply in some architectures and in such cases heap is always preferable. The automatic releasing of stack memory is its one big advantage, but there are problems with jumping into scope using `goto` or `select`. It is generally more difficult to detect a memory allocation error on the stack than the heap and provoking a stack overflow generally results in your program being terminated. With all of this said, it is worth pointing out that these same problems occur if you create a fixed-size `auto` array within a block and this is common practice.

At the end of the day it is simply the non-standard nature of the function that makes its use non-ideal.

Dive 13

First Class Functions

My work was fairly theoretical. It was in recursive function theory. And in particular, hierarchies of functions in terms of computational complexity. I got involved in real computers and programming mainly by being - well, I was interested even as I came to graduate school.

Dennis Ritchie

C is a language built on functions. No, it isn't a functional programming language, that would be a step far too far. It has functions like other languages before it had subroutines and procedures. C's functions are designed to be a way to break a large program up into small chunks. C was just another language that emphasized "modular" programming back when it was introduced.

Today, C's functions look a little under-powered as they lack any dynamic abilities that you will find in other current languages, but this can be seen as a strength. C's functions are simple and efficient and adding advanced features would dilute this rare quality.

The C Function

You probably know how to create a C function, but we need to make clear what is going on at a deeper level. A function is declared and defined in the usual way:

```
returnType name(parameterList){functionBody}
```

where {`functionBody`} is a block of code executed when the function is called with the parameter supplied.

This is a simple pattern and one that every C programmer learns right at the very start. There are also a number of features of C functions which are obvious, but often not made explicit.

A C function can only be created at file-level and the variable that you generally regard as the function's name is a file-level variable. That is, it is global as far as code in the same file or compilation unit goes. What is more, functions are true global variables, by default they are `extern`, and they can be referenced by code in other compilation units.

This is how C libraries work, they define functions which can be used by code in other files. If you want to limit the visibility of a function to its compilation unit, then mark it as `static`.

The fact that all functions are at file level implies that all the functions that you are going to use are defined at compile time and the lifetime of any function is the entire lifetime of the program.

If you think this is the only possibility you need to get out and meet some other languages. Modern dynamic languages – C#, Python, JavaScript, Kotlin, Rust and even Fortran 90 – allow functions to be defined within other functions. Such nested functions are local to the functions they are defined in and they can have lifetimes that depend on them being in scope. This is where some more sophisticated ideas, such as closure, come into play. If a nested function lives beyond the life of the function that contains it then it "captures" the local variables and they are available to the nested function. This is closure and exactly how the variables are made available depends on the type of closure implemented.

Other languages also allow functions to be modified by code – Python decorators, for example and this is something that was thought to be a very bad idea when C was being invented. Today, C functions cannot be modified at runtime without using machine- and compiler-specific features.

Partial Evaluation and Currying

What does C's approach to functions make difficult?

The most obvious things that are difficult, if not impossible, are partial evaluation and currying. Partial evaluation of a function returns a function with one or more parameters fixed. For example, if you have the function `sum(a,b)` then a partial application is:

```
add1(b) = sum(1,b);
```

where now `add1` is a function that always adds `1` to `b`. If you have a function of `n` parameters you can reduce it to a function of `m` parameters by applying `n-m` of the parameters.

Currying, a technique developed by and named after the American mathematician, Haskell Curry, is similar to partial evaluation, but it decomposes a multi-parameter function into a sequence of single parameter functions applied one after another:

```
sum(a,b,c) = add(a)(b)(c);
```

which means that `sum(a)` returns a function which can add `b` to the total and return a function which can add `c` to the total.

You cannot do either partial application or currying in C because you cannot create a new function at runtime, you cannot create nested functions and closures are not supported. This is a problem if you want to do functional programming in C. You also cannot implement object-oriented programming as you cannot create a bound function, i.e. a method – see Dive 14: Structs and Objects. Put simply, you cannot take a function:

```
myFunction(self, other parameters)
```

and use partial application to create a method:

```
myMethod(other parameters)
```

by setting `self` to reference the current instance:

```
myMethod = myFunction(instance, other parameters)
```

The key ideas are that C functions have file-level scope and are external by default, i.e. they are global variables. Functions also have a lifetime the same as the entire program and they can only be defined at compile time. Modifying a C function isn't possible without going beyond C standards.

Functions First

C's functions are simple and hence efficient, but their simplicity has a cost. C is a one-pass language in that it can be compiled in a single reading of the program text. This in turn implies that if you want to make use of something, a function say, it has to be declared before it is used. The reason is simple. If you use a function before it is declared the compiler knows nothing about it and cannot check if your use makes sense. This means that in C you have to define your functions before you use them and this means that they have to come in a particular order and all before the `main` function. Some C compilers will cope if they encounter a function that you haven't declared before use. They generally warn of "implicit" function declaration but this is not a good idea.

Putting functions before `main` isn't the natural way to read a program. You don't generally want to have to scan through all of the functions before you get to the main program and then have to backtrack to find the functions that are used. The natural way to do things is to read the main program and look up any functions that are used.

To provide a simpler program layout and organization, C allows you to declare variables, including functions, before they are defined.

Declaring a variable just informs the compiler of its type, no memory allocations have to be made at this point.

Defining a variable can be a separate step and it involves allocating memory and perhaps initializing it to a value.

We already know how to define a function:

```
returnType FunctionName(type parameter1, type parameter2 ...){
   statements that define the function
}
```

To declare a function we simply leave out the body, the block enclosed in curly brackets:

```
returnType FunctionName(type parameter1, type parameter2 …)
```

Naming the parameters is pointless as the compiler doesn't make use of them, so this can be simplified to:

```
returnType FunctionName(type, type  ...)
```

Some programmers prefer the brevity of just listing the types, some think that providing the parameter names informs the reader what the function does, especially so if a comment is used. For example:

```
int sum(int, int);
```

defines the sum function as taking two integers and returning an integer, but

```
int sum(int a, int b); // returns a+b
```

This works when the function is simple, but is less effective as the complexity and number of parameters increases. In most cases it is better to include a brief declaration and leave the comments for the definition.

Once you know about declaring functions, a C program can have a standard form:

```
function declarations
main(){...}
function definitions
```

and to make this even easier to read, it is conventional to put the function declarations into a header file.

```
#include declarations.h
main(){…}
function definitions
```

Things are a little more subtle than this in that the function definitions don't have to be in the same file/compilation unit. By default function declarations are assumed to be `extern`, i.e. external. This means that if you declare a function and do not define the function within the same file, the linker will look for the function within the libraries that have been specified.

What this means is that by default any function you declare and define in a file can be used from other files. In this sense every function is by default global to the entire project.

If you want to restrict the function to be "local" and defined in the same file prefix the declaration with `static`:

```
static int sum(int, int);
```

this means that the `sum` function has to be defined in the same file and it is not available for use in other compilation units – it is internal to the file. Use `static` in a function declaration when you want to avoid name clashes with functions in any libraries you might be using.

What about variables that are not function pointers, do you need to declare them?

In most cases you can declare and define a variable in the usual way:

```
int myInteger;
```

This allocates the memory needed as well as telling the compiler the type of the variable. Of course, variables have to be declared before they are used, but where they are declared modifies where the memory needed is allocated from. The only example of declaring a variable without defining it is when you declare the variable as `extern`:

```
extern int myInteger;
```

This informs the compiler that `myInteger` is an `int`, but doesn't allocate any storage for it as it is assumed that this is done in another compilation unit. That is `myInteger` is a variable in another compilation unit (file).

You can declare a variable as many times as necessary, but you can only define it once.

Function Pointers and Type

The concept of a "first-class function" is a relatively recent one and C's functions were thought up well before it became an issue. The basic idea is that a first-class function is one that is treated the same as other objects in the language. Where "objects" can mean simple things like ints or floats or bigger things like class-derived instances of complex data structures. However, the key feature when a function is "first-class" is the ability to pass it as a parameter to another function and C has allowed this from the start.

A function name is a pointer to the starting address of the function. If you write:

```
printf("Address = %X",&myFunction);
```

or

```
printf("Address = %X",myFunction);
```

you will see the address that `myFunction` starts at. In other words the `&` is optional when dealing with a function name.

What this means is that you can pass a function, using its name, to another function and then call it from within that function. The only problem we have is, what type is a function? To define a parameter as accepting a particular function, we need to be able to define the type of a function.

The type of a function is simply a pointer to a function.

This is easy in principle, but slightly difficult to read:

function return type (**function name*)(*parameter types*);

So, for example:

```
int (*myFunction)(int,int);
```

defines a function that returns an `int` and accepts two ints as parameters. Notice that this is similar to the usual function declaration:

```
int myFunction(int a, int b);
```

When you create a type for a function you can include parameter names if you want to. Notice that the parentheses around the function name are necessary as:

```
int *myFunction(int a, int b)
```

is a function that returns a pointer to `int`.

To be clear:

```
int (*myFunction)(int,int);
```

defines the variable `myFunction` to be a pointer to a function that accepts two ints and returns an `int`. Notice that the name of the pointer isn't at the end of the declaration as it usually is – this makes reading such declarations more difficult.

For example:

```
int (*myFunctionPointer) (int,int);
myFunctionPointer=sum;
int result=myFunctionPointer(1,2);
```

The function pointer can be set to reference the function `sum` as it has the same signature. Once you have a function pointer, you can use it to call the function in the normal way.

As function types are complicated to read, it is usually better to define a type using `typedef`. In this case the name supplied is used as the name of the type:

```
typedef int (*myFunctype) (int,int) ;
myFunctype myFunctionPointer;
myFunctionPointer=sum;
int result3=myFunctionPointer(1,2);
```

Notice that the name of the type defined by the `typedef` isn't at the end of the statement which is where you usually look for it. However the declaration of the function pointer is now more usual in that the name of the variable is at the end rather than in the middle.

Notice that any function that matches the type in terms of the number and type of its parameters and return type can be considered to be of that type. You can even define multiple identical, but differently-named, types and they will all be treated as the same.

You cannot use a raw function pointer to define a function. That is you cannot write:

```
int (*myFunction)(int a,int b)
{
 return a+b;
}
```

You have to define a function and then assign it to the function pointer.

Function Parameters and Return Types

You can use a function pointer as a parameter in a function so that you can pass a function into another function as an argument.

For example:

```
void myFunction(int(*myFuncParm)(int,int)){
  printf("%d\n",myFuncParm(1,2));
}
```

and again notice that the name of the parameter, `myFuncParm`, is not at the end of the type specification.

This can be made much easier to read with the help of a `typedef`:

```
typedef int (*myFunctype) (int,int);

void myFunction(myFunctype myFunction){
 printf("%d\n",myFunction(1,2));
}
```

Defining the type of the return as a function pointer is messy, but it is rarely needed as C functions are not dynamic. That is, a function cannot create a function and return it. What this means is that a function that returns a function can only return a function that already exists and is in scope.

To define the return type of a function as a function you have to supply the type in front of the function definition as usual but in this case the type includes the name of the function.

return type (**name of function*(*parameters1*)) (*parameters2*)

where the function returned has type:

return type (*)(*parameters2*);

and the function being defined is:

name of function(*parameters1*)

For example:

```
int sum(int a, int b){
 return a+b;
}

int (*myFunction(void)) (int,int)
{
 return sum;
}

…
int (*summer)(int,int) =myFunction();
printf("%d\n",summer(2,3));
```

In this case the name of the function, `myFunction`, is within the type declaration and it is defined to return a function pointer that takes `int,int` and returns `int`. Notice that the function definition includes its parameters – in this case `void`. If it accepted `float` and `int` parameters, the declaration would be:

```
int (*myFunction(float,int)) (int,int)
{
 return sum;
}
```

You can even define a function that accepts a function and returns a function:

```
int (*myFunction(int (*f)(int,int))) (int,int)
{
 return f;
}

int (*summer)(int,int) =myFunction(sum2);
printf("%d\n",summer(2,3));
```

This looks, and is, difficult to read, but it is perfectly logical. If you define a function type then the same declaration looks much simpler:

```
typedef int (*myFunctype) (int,int) ;

myFunctype1 myFunction(myFunctype1 f){
    return f;
}
```

and is used in the same way. The only disadvantage of this approach is that you need to define a function type for each function you want to use as a parameter or `return`. Of course, given the static nature of C's functions it is difficult to see why you would want to pass in a function and then return it, but it can be done.

Parameters and Arguments

We have looked at how functions can be used as parameters, but it is worth looking more generally at the way parameters behave.

Parameters act like local variables initialized when the function is called. For example:

```
void myFunction(int myParam1, char myParam2){
```

called as:

```
myFunction(42,'H');
```

can be thought of as:

```
void myFunction(int myParam1, char myParam2){
  myParam1 = 42;
  myParam2 = 'H';
```

This is also quite close to what really happens. When you call a function the expressions that you specify for each parameter, the arguments, are evaluated and each value is pushed onto the stack along with the return address before the function is called – see Dive 8. When the function starts to execute, the parameters are on the stack along with any other local variables the function declares. This description is idealized in that many compilers attempt to optimize the program by placing the first few parameters into registers to speed things up. However, the stack idea is still a useful way of thinking about things.

In this sense parameters behave like pre-initialized locals.

If you want to pass something in by reference then you have to pass an address. We are all familiar with the difference. If you "pass by value" then any changes to the parameter do not affect the variable passed in:

```
void myFunction3(int a){
    a = 42;
}
int a = 43;
myFunction(a);
```

After the function, `a` in the calling program still has `43` stored. The fact that the parameter has the same name is irrelevant. A "pass by value" doesn't always need you to use a variable as the argument. For example:

```
myFunction(21*2);
```

is perfectly acceptable. It really is better to think of the argument as always being an expression which is evaluated and passed into the function.

Passing a reference works in exactly the same way. What changes is the way you treat the parameter. For example:

```
void myFunction(int *a){
    *a = 42;
}
```

Now we are passing a pointer to an `int` and using the dereference operator in the assignment means that `42` is stored in whatever `a` references. Now if you call the function:

```
myFunction(&a);
```

passing the address of `a` in the calling program, then the function changes the value stored in `a` in the calling program. So C permits "pass by reference", but it is criticized because it is such a manual operation that it invites errors. If a parameter is to be passed by reference, it has to be a pointer, e.g. `*a`, and within the function it has to be dereferenced, i.e. you need to use `*a`. What is more, you need to remember to call the function with the address operator, e.g. `&a`. If you accidentally forget a `*` or a `&` the function doesn't do what you think it does.

Other languages automate the process, for example C# uses:

```
myFunction(byref a)
```

to define a reference parameter. When you call `myFunction` it automatically takes the address of the variable and automatically dereferences it within the function. This is more foolproof, but less flexible. The same idea is the basis of the C++ extension to C, passing by reference. In this case, if you add `&` in front of a parameter it is passed by reference and its address is obtained and dereferenced automatically. In C++ our previous example can be written:

```
void myFunction(int &a){
 a=42;
}
myFunction(a);
```

Why pass by reference?

There are only two reasons for doing this. The first is that you want to use a parameter as a way of getting a result out of a function. For example:

```
void sum(int a, int b, int *c){
    *c=a+b;
}
```

which returns the result of the sum in `c`:

```
int result;
sum(3,4,&result);
```

In this case you could have returned the result in the usual way, but if you have multiple items to return then reference parameters can be used. However, using a `struct` is arguably better, see later.

The second is that you want to avoid having to make a copy of a large data structure. This is the reason that arrays are automatically passed by reference as an array name "decays" to a pointer when used in this way. For example:

```
void myFunction(int *a)
{
    *(a + 2) = 42;
    a[2] = 42;
}
int myArray[5] = {0,1,2,3,4};
myFunction(myArray);
```

and the assignment to `a[2]` changes the array in the calling program. The only danger for the beginner is not noticing that arrays are pointers and hence passed by reference.

What if you want to pass an array by value? There is no straightforward way of doing this, but you can pass anything by value if you wrap it in a `struct`, for example:

```
typedef struct{
   int myArray[5];
} arrayType;

void myFunction(arrayType array){
  array.myArray[2]=42;
}
arrayType myArray={{0,1,2,3,4}};
myFunction(myArray);
```

If you try this out you will find that `myArray` in the calling program is unchanged by the function. The cost of this transformation is the need to use `arrayType` for the type of the array and the qualified name to access the `myArray` field. You also have to set the size of the array in `arrayType` which makes it difficult to reuse. You can simplify the code in the function by aliasing `array.myArray` to a pointer:

```
int *array2 = array.myArray;
array2[3]=43;
```

You can alias `array.myArray` in the same way in the calling function to make it look more like a standard array, but you have to pass the `struct` to the function to invoke the pass by value mechanism.

Calling Conventions

So far we have taken the simplified view that arguments are pushed onto the stack for use by the function and the return value is also passed back on the stack. In practice, exactly how this happens is important and it gives rise to the idea of the calling convention, i.e. the rules for passing parameter values.

The simple-minded view is that the calling convention doesn't matter as your program will be compiled by a compiler using a single calling convention and, as this just works, the details hardly matter. In practice, however, your program is going to need to call functions provided in libraries that are already compiled. For example, under Windows you can call functions in pre-compiled DLLs, Dynamic Link Libraries. In Linux you can call functions defined in shared or `.so` libraries. Even if you are not using libraries, you most likely want to call functions provided by the operating system. In any of these cases it is important that both the calling code and the callee make use of the same calling convention.

Calling conventions specify the order in which arguments are pushed onto the stack and, to improve performance, they may specify an order in which registers are used in place of the stack. For example, Microsoft defined the `fastcall` convention where the first two arguments are passed using the ECX and EDX registers and any additional are pushed onto the stack from right to left. Of course, many functions only have one or two parameters and so these will be faster as the stack isn't used at all.

Mostly you can ignore the calling convention as the compiler will use the one most appropriate for the system you are using, but sometimes foreign code requires a specific calling convention and then you need to tell the compiler what to use. For example, in 32-bit Windows systems, functions use the `stdcall` convention, but most C programs use the `cdecl` calling convention. Another reason for selecting a particular calling convention is to optimize the code. For example, Microsoft introduced the `vectorcall` convention which extended the `fastcall` convention to allow larger data structures to be passed in 128- and 256-bit registers, significantly speeding up graphics operations.

The idea of a calling convention leads on to the bigger concept of an Application Binary Interface, ABI. This not only defines a calling convention, but also how data is laid out in memory so that pre-compiled binary code can use other pre-compiled binary code. If an ABI changes then generally all of the code that works together has to be recompiled.

Ignoring Parameters

Although mentioned in Dive 7 on `void`, it is worth repeating that the declaration:

```
void function();
```

is not a function with no parameters. It is a function with an unknown number of parameters. That is, following this declaration you can define a function with any number of parameters including none:

```
void function(int a, int b){};
```

To specify that a function takes no parameters you need to use:

```
void function(void);
```

After this declaration you can only define a function with no parameters - anything else generates an error.

If you specify a given number of parameters, then you have no choice but to supply them all when calling the function. This raises the question of what to do when a particular call doesn't need some of the parameters. In other words, how do you implement optional parameters in C? The usual answer is that you have to pass a value that indicates that the parameter is a dummy. The most common choice of value is `NULL`, which seems sensible until you think about what value is actually passed.

For example, if you have a function which finds the maximum of three values:

```
int max(int a, int b, int c){
 if(a>=b && a>=c) return a;
 if(b>=a && b>=c) return b;
 return c;
}
```

Then `max(-1,-2,-3)` will return `-1` as this is the largest value. Now suppose you want to use `max` to find the maximum of just two values. You might be tempted to use:

```
max(-1,-2,NULL);
```

but this results in `max` being returned as `0` instead of `-1`. The reason is, of course, that `NULL` is just a macro that expands to `0` and the call is:

```
max(-1,-2,0)
```

which results in `0` being the maximum.

You might think that this can be fixed by testing for `c` being `NULL`:

```
if(c = NULL) …
```

but this simply rules out setting `c` to a valid zero value.

In this case the only safe way of ignoring a parameter is to set it to the same value as either `a` or `b`:

```
max(-1,-2,-1);
```

returns `-1` as the max and clearly setting `c` to the same value as `a` or `b` cannot change the result.

Ignoring a parameter is in general difficult as you need a value to use as a sentinel that isn't valid in normal use and zero disguised as `NULL` isn't usually a good choice.

You can usually avoid the problem by including a flag parameter which indicates if the optional parameter is used:

```
int max(int a, int b, int c, int number){
  if(number==2)c=a;
  if(a>=b && a>=c) return a;
  if(b>=a && b>=c) return b;
  return c;
}
```

and now it is safe to set `c` to `NULL`:

```
max(-1,-2,NULL,2);
```

In general, arranging for an optional parameter is something that needs a lot of thought. Perhaps the one exception is when the passed value is an address:

```
int max(int *a, int *b, int *c){
 if(*c==NULL){
       if(*a>=*c) return *a;
       else return c;
 if(*a>=*b && *a>=*c) return a;
 if(*b>=*a && *b>=*c) return b;
 return c;
}
```

Now you can call the function using `NULL` as an indicator that `c` isn't used:

```
max(&x,&y,NULL);
```

as zero is almost never a valid address.

In short `NULL` is useful when passing pointers, but not so much for other types.

Variadic

Perhaps the ultimate in ignoring parameters is the variadic function. "Variadic" is derived from "vari", variable, and "adic" meaning the number of parameters. A variadic function can accept a variable number of parameters. This sounds good, but C's implementation is fairly primitive compared to what other languages can do. You have already been using a variadic function for most of your C programming career – the `printf` function is variadic.

To specify a variadic function you simply end the parameter list with three dots, …, an ellipsis. The C standard states that the function has to have at least one explicit parameter and this rules out pure variadic functions. A variadic function has to be declared using `...` before you can use it. Calling a variadic function is just like calling any function but you can now use as many parameters of any type, apart from the first few normal parameters. For example:

```
int max(int n,...);
```

declares a variadic function and:

```
int result=max(3,1,2,3);
```

calls it. In this case the first parameter is being used to specify the number of optional parameters. This isn't essential, but it is common practice.

To process the optional parameters you have to make use of some predefined macros in the `stdarg.h` file. The idea is that as parameters are passed on the stack there is no need to restrict the number of parameters that are pushed onto the stack when the function is called. The variadic function can use a macro to get a pointer to the stack and a macro to retrieve each argument in turn.

The macro:

```
va_start(args, argN)
```

sets the variable `args` of type `va_list` to reference the first variable parameter and this is located on the stack after `argN`, the last fixed parameter. For example:

```
va_list args;
va_start(args,n);
```

sets `args` to reference the first optional parameter which is located on the stack after the *n* parameter.

The macro:

```
va _args(args,type);
```

retrieves the next argument from the stack. The `type` parameter is used to specify how far to move the reference on so that it references the next parameter. For example:

```
int a=va_args(args,int);
```

retrieves an `int` argument and moves `args` on to reference the next argument if there is one. Notice that you can read as many or as few arguments as you like. If you read beyond the number of arguments specified then whatever is in the memory is returned.

When you are finished with the reference to the stack you can call:

```
va_end(args);
```

to free up the pointer, but in practice this usually is no operation.

You can pass `args` on to another function if you want to. You can also obtain multiple argument pointers and use each one independently to access the arguments. You can also ignore all or some of the optional arguments. There is also a copy function, added in C99, that you can use to make a copy of the pointer:

```
va_copy(dest,src);
```

This allows you to mark where you got to in the argument list and continue at a later time. In versions before C99, you can generally make a copy by direct assignment.

Notice that the idea that the arguments are all stored on the stack may be a simplification as many compilers will use registers for the first few parameters in a function – but it is a useful abstraction.

The big problem with using variadic functions is that the whole process is error-prone. You can get the number of arguments wrong and you can get their types wrong and the compiler will not inform you that there is a problem, but you will be working with mangled data. You can use the usual two ways of fixing the number of parameters – supply the number as the first fixed parameter or use a sentinel value.

For example, to find the maximum of a variable number of parameters:

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

int max(int n, ...)
{
    va_list args;
    va_start(args, n);
    int maxvalue=va_arg(args,int);
    int temp;
    for(int i=1;i<n;i++){
        temp = va_arg(args,int);
        if(maxvalue<temp) maxvalue = temp;
    }
    return maxvalue;
}
```

Notice that there is no check that n is sensible and if n is specified incorrectly the function will read memory locations that it hasn't set.

You can call this function using something like:

```
  printf("%d\n",max(4,3,4,2,9));
```

You could use a sentinel value to mark the end of the parameters, but what value should you use?

Another approach is typified by `printf`. The format string specifies a parameter and a type. For example a format string `"%d%d%d"` specifies that three arguments, each an `int`, should be supplied:

```
printf("%d %d %d",a,b,c);
```

Notice that this implies that `printf` isn't entirely safe in that you can supply a smaller number of arguments than the format string specifies.

Return

As we have already discovered, a function can return a value, again using the stack, and this is used as if it was the result of evaluating an expression with the value being pushed onto the stack for the calling program to use.

This is "call and return by value". When you call a function the parameters are copied onto the stack and the return value is copied onto the stack.

Notice that a function that doesn't return a value, i.e. its return type is `void`, doesn't have to have a return statement. Equally, if a function does return a value, the calling program can perfectly well ignore it and essentially discard it. A function that doesn't return a value corresponds to what would be called a "procedure" or a "subroutine".

Using pass by value has implications for memory safety. For example:

```
typedef struct {
 int myInt;
 } myStructType;

myStructType myFunction(void){
  myStructType myStruct={42};
  return myStruct;
}

main program
…
myStructType myReturnStruct= myFunction();
```

This is perfectly fine as the `myStruct` is allocated on the stack as an `auto` variable and when the function ends it is copied byte-by-byte to the newly created `myReturnStruct`. However, make a small change to the program and things go wrong:

```
myStructType *myFunction(void)
{
    myStructType myStruct = {42};
    return &myStruct;
}
```

Now we are returning a pointer to the struct, we are returning it by reference. If you try to make use of this:

```
myStructType *myReturnStruct = myFunction();
```

then we have a pointer to a struct that was created on the stack as a local variable and this now no longer exists as the function has exited. The result is usually a runtime error, but it may be slow in coming as the memory might be valid for some time.

The key principle is:

- Never return a pointer to a local variable.

Most compilers will warn you if you try to do this, but beginners often misunderstand that warning because everything seems to work – at least for a while.

Functions and Big Data

If you can't return a pointer to a local variable and passing by value isn't efficient, how should a C function return a large data structure? There are two broad approaches – import it or export it.

The "import it" approach relies on the calling function to provide the data structure. For example:

```
void myFunction(int *array, int N){
    for(int i=0;i<N;i++){
        array[i]=i;
    }
}
…
int *myArray = (int*) malloc(sizeof(int)*10);
myFunction(myArray,10);
```

`myFunction` needs an array to work and the calling program provides this. The disadvantage of this method is that the calling program has to know the size of data structure needed – how is it to know that `N = 10` is appropriate if the array is being used as a buffer for data read in from the outside world? The huge advantage is that the calling program is responsible, and knows very clearly that it is responsible, for disposing of the memory allocated. If the data structure is an `auto` then this happens without any effort. If the data structure is allocated on the heap then it has to be freed when finished with.

The “export it” approach leaves the function to create the data structure and return a pointer to it:

```
int *myFunction(void){
    int *myArray = (int*) malloc(sizeof(int)*10);
        for(int i=0;i<10;i++){
        myArray[i]=i;
    }
    return myArray;
}
int *myArray = myFunction();
```

The advantage is that the function can decide how big a data structure is needed, but it has to pass this back to the calling function. The disadvantage is that the user of the function may not have any idea that it is responsible for freeing the data structure. It might even be that the function returns the same data structure on each call leaving multiple callees jointly responsible for the lifetime of the structure. In most cases the best alternative is to provide a management function to complement `myFunction` which will dispose of the structure and that can be called multiple times without causing an error.

Returning More Than One Result

A C function can only return a single value and this can be a problem. If you need to return multiple values, a single value return may not seem up to the task – it is but it isn’t obvious.

A very common approach to the problem is to use parameters to return results. We have already seen that passing a pointer is effectively pass by reference and as such the function can change values in the calling program. Consider a function that needs to return a `max` and a `min` value. We could implement this as a function with two pointer parameters:

```
void maxmin(int *max, int *min){
    *max = 100;
    *min = 0;
}
int mymax;
int mymin;
maxmin(&mymax,&mymin);
```

As always, the problem is to remember to pass in the address of the variables being used for the results. Usually such functions are organized so that the “in” parameters are listed before any “out” parameters.

An alternative, and arguably a better way is to use a `struct` to package any results:

```
typedef  struct {
        int max;
        int min;
    }tuple;

tuple maxmin2(void){
    tuple result;
    result.max=100;
    result.min=0;
    return result;
}
...
tuple result2=maxmin2();
```

The disadvantage of this approach is that you have to declare a `struct` suitable for returning the results and this is often difficult from the point of view of scope and finding a reasonable name for it. The advantage is that the results are returned as they should be by a `return` statement which makes the function easier to understand.

In practice you will encounter the use of “out” parameters more often than structs. The reason is probably that “out” parameters can be introduced in an ad-hoc manner whereas using structs requires some forethought and planning.

Inline Functions

Functions are the best way to organize your program – a function for every identifiable task. Some programmers worry that having so many functions is inefficient. The overhead is the time taken to call the function and the time for the return. In practice, this overhead is usually well worth the improvement in program maintainability. However, sometimes a function does so little that it seems that it might be better to code it each time it is needed. Inline functions give you the best of both worlds, but they were only introduced in C99.

Any function that is marked as `inline` has its call expanded as if its body had been written in place of the call. However, things aren’t quite as simple as this.

The `inline` qualifier is only a hint to the compiler to inline the function. If the compiler decides not to then you will get an error from the linker as there is no non-inline version of the function defined. To avoid this you can mark a function as `static`. For example:

```
static inline int add(int a, int b){
    return a+b;
}
int main(int argc, char** argv) {
    int c= add(2,3);
    printf("%d \n", c);
    return (EXIT_SUCCESS);
}
```

It is usually said that you don't need to use `inline` with modern compilers as they will optimize your code without the help of your suggestions.

C Functions Considered

As already repeatedly mentioned, C functions are fast, efficient and simple and for a low-level language this is exactly what you need. But this simplicity means that there are features that a programmer familiar with other languages may find lacking.

C functions are static in the sense that they have to be fully defined at compile time and they cannot be changed without the use of clever, platform-dependent tricks. The name of a function is a pointer to the start of the function and it is a project-global variable unless you restrict it to the current file using the `static` keyword. This implies that the lifetime of a C function is the same as the entire program.

C functions only have positional parameters - no keyword parameters, optional parameters, default values or optional parameters are possible. You can create functions that accept a variable number of parameters, but determining how many parameters are in use and their type is difficult.

As C functions cannot be defined within other functions, i.e. local functions aren't supported, there is no support for closure, partial application or currying.

Dive 14

Structs and Objects

"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

Edsger Dijkstra

Apart from the array, the struct is the only other sophisticated data structure that C provides. As things turn out, the array and the struct are enough for most things. At the most basic level, the struct is the C implementation of the data record, but the data record is also where object-oriented programming starts. This might seem surprising, but it is true that in most languages there is little difference between an object and a record. To understand C's approach, the all purpose struct, let's start with the basics.

The Struct

You were probably introduced to the `struct`, short for "structure", as a sort of array that allows different types as elements. For example:

```
struct{
       int myfield1;
       float myfield2;
} myStruct;
```

This creates a `struct` with two fields – one an `int` and the other a `float`. These fields are grouped into the structure in the same way as elements are in an array – except that they are of different types. Of course, you access the fields using the familiar dot notation:

```
myStruct.myfield1 = 42;
myStruct.myfield2 = 4.2;
```

This declaration is of the standard form:

`type variable``;`

where *`type`* is the complete `struct` definition and the *`variable`* is the name at the end of the type.

For example:

```
struct{
       int myfield1;
       float myfield2;
}
```

and the variable is `myStruct`. This form of struct type is called an "anonymous struct" because the type itself lacks a name other than `struct`.

You can take this a little further by giving the type a name, a tag. This allows you to reuse it. For example:

```
struct myStructType{
       int myfield1;
       float myfield2;
};
struct myStructType myStruct;
```

You can use `struct myStructType` as often as you like to create as many structs of the type as you need.

The full name of the type is `struct myStructType` and you can get rid of the `struct` part of the name by using a `typdef`

```
typedef struct {
       int myfield1;
       float myfield2;
}myStructType;

myStructType myStruct;
```

Now the anonymous struct has been given the name `myStructType` and it can be used in the standard way to declare as many variables as you like. Introduced like this, the C struct declaration doesn't seem in the least bit illogical and it is quite obvious why there are different ways of declaring a struct.

The main use of the anonymous `struct` is to declare struct fields in other structs. For example:

```
struct{
    struct {int a,b;} myStruct1;
    float value;
}myStruct2;

myStruct2.value=4.2;
myStruct2.myStruct1.a=42;
```

There are many other struct idioms that make using structs more compact.

Value Semantics

Structs, unlike arrays, apply value semantics. What this means is that the variable associated with the struct behaves as if the struct was its value. Contrast this to the variable associated with an array behaving as if it was a reference i.e. a pointer to the array. You cannot assign one array to another because the variables only hold references to arrays, but you can assign an array to a pointer:

```
int myArray1[10];

myArray1[0] = 42;
int *myArray2 = myArray1;
myArray2[0] = 43;
```

In this case, changing `myArray2[0]` also changes `myArray1[0]` as there is only one array that both variables reference. This is reference semantics at work.

Compare this to:

```
myStructType myStruct1;

myStruct1.myfield1=42;
myStructType myStruct2=myStruct1;
myStruct2.myfield1=43;
```

In this case, the contents of `myStruct1` are copied to the new `myStruct2`. If you now change `myStruct2` it has no effect on `myStruct1`. This is value semantics at work.

Put simply, unless you explicitly use a pointer, any operations involving a `struct` generally involve making a copy of all of the struct's data. When you pass an array to a function then it is a pointer which is passed and the function works with the same copy of the array as the calling program. When you pass a struct to a function, a copy of the struct is made on the stack for the function to work with in isolation, i.e. it is passed by value.

What is more a struct is returned by value. That is, if you create a `struct` within a function as an `auto` then it is created on the stack. If it is returned to the calling program then its value is copied to the calling program's stack and it is safe to use, even though the original is destroyed. This is not the case with arrays. For more about functions, value and reference see Dive 13.

It comes as a surprise to many programmers that structs are value types for the simple reason that a struct can be large and making copies of large amounts of data isn't very efficient. The solution to this problem is to make use of a pointer to a struct and pass this instead. As a C program becomes more sophisticated, the tendency to use pointer to struct makes a struct look more like a reference type.

It's Just a Block of Memory

At this point it is worth starting to think of the `struct` type as defining how a block of memory is divided up into subtypes. For example, when you write:

```
struct myStructType{
       int myfield1;
       float myfield2;
};
```

you are saying that any block of memory that this `struct` is applied to will have `sizeof(int)` bytes as an `int` followed by `sizeof(float)` bytes as a `float`. When you create such a struct, the compiler allocates the correct amount of memory to store the field. This means that `struct` has to have a known size at compile time. What this means in practice is that you can't use a VLA in a struct and any types you use have to be fully defined at compile time, although see later for how to include a dynamic array.

The memory for the `struct` is allocated according to where the declaration is. If it is at file-level then the memory that the `struct` needs is compiled into the program. It if is within a function, then it is allocated on the stack. You can also create a `struct` on the heap by simply allocating the memory and casting to the type:

```
struct myStructType *myStruct=
       (struct myStructType *) malloc(sizeof(struct myStructType));
```

All of the usual lifetime rules apply to structs.

Notice that a pointer to a struct can be used in the usual way:

```
(*myStruct).myfield1 = 42;
```

but you do need the parentheses as the . operator has a higher priority than the * operator. To make this easier you can also use the -> operator to dereference and access the field:

```
myStruct->myfield1 = 42;
```

No dereferencing or parentheses needed.

Padding

The only tricky part is that a struct may actually use a larger block of memory than you might expect. The reason is "padding" due to the need for memory accesses to be aligned to word boundaries. For example in a 32-bit machine the address of a 32-bit `int` would have to be a multiple of four because the memory is divided into 4-byte words, accessible in a single read/write.

Some machines demand that memory accesses are aligned, others allow unaligned access, but warn that it is slower. If alignment is optional, some

compilers have a facility that lets you decide if you want to pack the struct for minimum space or include padding to ensure that fields are aligned correctly for speed. What all this means is that you cannot simply assume that a struct takes the amount of space suggested by how much space its fields take.

For a 64-bit machine using a 64-bit word, a `struct` will take at least eight bytes.

For example:

```
struct test {
    int i;
    char c;
}test;
printf("%d \n", sizeof (int));
printf("%d \n", sizeof (char));
printf("%d \n", sizeof (struct test));
```

will print `4`, `1` and `8` on a 64-bit Intel/Arm processor based machine. So although the `struct` should have taken just five bytes it actually takes eight to pad it out to a full 64-bit word.

Notice that the padding is added at the end or in between fields. The C standard ensures that no padding is added to the start of a `struct` and this means you can rely on the fact that the address of the struct is the address of the first field but, of course this first address will be on a word boundary.

If you want to pack a `struct` to take the smallest amount of memory then a rule of thumb is to allocate fields in the order of size starting with the largest.

The fact that structs are packed makes many things that you might think easy more difficult. For example, you cannot rely on pointer arithmetic to pick out fields because apart from the first field padding may move the start of a field from where you expect it to be.

Type Punning

Apart from the issue of padding, you can treat a `struct` as just a block of memory in the sense that you can cast it into other forms. This is an example of “type punning” where the bytes of one type are interpreted as the bytes of another. This is a very standard operation when C is used in a low-level system role and yet the whole subject is fraught with confusion. Some programmers are so convinced that type punning is a bad idea that they state that it should never be used. The fact of the matter is that it is used a lot and if it really was to be removed, a lot of code would have to be rewritten and some would be next to impossible to implement in an efficient way.

The simplest example is where a function accepts a `struct` as a parameter, but the `struct` has been extended over the years of development. Suppose it started out as:

```
struct myStruct{
  int a;
  int b;
};
```

and the function is:

```
void myFunction(struct myStruct *myPointer){
       printf("%d\n", myPointer->a);
       printf("%d\n", myPointer->b);
}
```

Now suppose that later on the struct is extended to include a new field essential to the further development of the project:

```
struct myStructEx{
   int a;
   int b;
   int c;
};
```

The good news is that new functions can make use of the new `struct` and the old functions can carry on treating the new `struct` like the old using a cast:

```
myFunction((struct myStruct *) myPointer);
```

where `myPointer` is now a pointer to `myStructEx`. Obviously `myFunction` knows nothing about the extended structure and so will only use the fields `a` and `b`. This is the struct analog of subclassing – `myStructEx` derives from `myStruct` in exactly the same way as a subclass derives from a base class. Cast to the "base" struct is an example of type punning where a block of memory is treated as different types.

You can take the "inheritance" idea a step further by including the base `struct` in the derived `struct`:

```
struct myStruct
{
       int a;
       int b;
};
struct myStructEx
{
       struct myStruct base;
       int c;
};
```

Now any new functions passed `myStructEx` have to refer to the base `struct` as

```
myPointer->base.a;
```

but if you cast `myStructEx` to `myStruct` the original still works:

```
(myStruct *)myPointer->a
```

If you think that this sort of thing is rare and not something you are likely to use or encounter, it is worth pointing out that CPython is based on this way of implementing classes and objects. However, it is important to know that nearly all type punning breaks the strict aliasing rule. This basically states that a block of memory should only be referenced by pointers to compatible types, see Dive 16. In other words, one of the most used of the advanced features of C is usually illegal according to the recent C standards. This is, of course, nonsense. Use type punning wherever it makes sense and if necessary configure the compiler to swallow it without demur, the alternatives are much worse.

Notice that you do have to be careful about padding being different if you change the type of a struct. For example, if you have a `struct` like:

```
struct test1 {
     char c;
      int i;
};
```

and you pun it with:

```
struct test2{
    char c[1];
    char i[4];
};
```

then you are making the assumption that the layout is one byte for the `char` followed immediately by four bytes for the `int`. If you try this out on a 64-bit machine you will find that `c[0]` is the same as `c`, but `i[4]` doesn't represent the integer. The reason is for the problem is that the `struct` is packed so that the `int` starts on a word boundary. So the layout is one byte for the `char` and then three bytes of padding to make the four bytes of the `int` start on a word boundary. A correct pun is:

```
struct test2{
    char c[4];
    char i[4];
};
```

with `c` corresponding to `c[0]` and the `int` corresponding to `i[0]` to `i[3]`. Clearly this is machine-dependent and there is no way you can gain access to the bytes that make up the `int` in a machine-independent way.

Structs As Objects

`Structs` are so close to being objects that we don't need to do much to make them even more like objects. An object has fields that are data or functions. The functions are the object's methods and they are used to manipulate the data in the object.

For example, a list object may have fields that store data and fields which specify methods such as sort. To use the method you would write:

```
myObject.sort();
```

and you would expect the sort method to perform a sort operation on the data in `myObject`. Notice that the non-object version of this is to simply have a sort function that accepts the data to be sorted as a parameter:

```
sort(myObject)
```

Can it really be that the only difference between object- and non-object-oriented programming is the switch from `sort(myObject)` to `myObject.sort()`? Obviously this isn't all there is to it, but it is a large part of the idea. The key idea is that `myObject` is associated with a `sort` function that "knows" how to sort it into order.

Usually when a function like `sort(myObject)` is implemented as part of an object they are called methods and the difference is that the sort function is bound to the object. This binding is usually performed by providing an additional local variable inside the function, often called self or this, which references the object. The method can work with the object using self or this to identify the instance of the object that is calling the function.

In C, functions can have pointers that reference them, see Dive 13, and as such they can be fields in a struct. For example:

```
struct MyClass
{
    int data[10];
    void (*sort)(struct MyClass*);
    void (*display)(struct MyClass*);
};
```

This defines a `struct` that has a data field and two functions which are intended to sort and display the data. The only part that is difficult in this is the specification of the type of the function pointer, see Dive 13 for more examples.

The methods are declared as standard functions:

```
void Sort(struct MyClass *self)
{
    printf("sort called\n");
}

void Display(struct MyClass *self)
{

    for (int i = 0; i < 10; i++)
    {
        printf("%d\n", self->data[i]);
    }
}
```

To make use of this we need to create an instance of the class:

```
int main(int argc, char **argv)
{
    struct MyClass myObject = {{0,1,2,3,4,5,6,7,8,9},
                               &Sort,
                               &Display};

    myObject.sort(&myObject);
    myObject.display(&myObject);;
}
```

This sets up an instance of the `struct` and initializes it so that it has an array to sort and two functions to do the jobs. The functions can be called in the usual way, but notice that they are functions as we have to pass a pointer to the instance of the `struct` as the first parameter. There is no way that the functions can "work out" which instance of the `struct` to sort or in the jargon the functions are not automatically bound to the instance they operate on.

In C functions cannot be "attached" to a struct in such a way that the function "knows" which struct it was called by. To do so we would need to generate a new "bound" function, i.e. a method by partial application that eliminated `self` from the function call but provided it automatically, see Dive 13 and partial application.

It can be done, as C is so low-level anything can be done, but only by knowing how the function call and return is implemented and this is highly machine-dependent.

We can introduce a constructor function to make the whole think look more like an object:

```
struct MyClass *new_MyClass(int data[10])
{
    struct MyClass *instance = malloc(sizeof(struct MyClass));
    memcpy(instance->data, data, sizeof(int) * 10);
    instance->self = instance;
    instance->sort = Sort;
    instance->display = Display;
    return instance;
}
```

The constructor allows us to initialize the object with an array of data which is copied into the data field of the struct using `memcpy` – this is very efficient. Now we can construct an object and use it more naturally:

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

    int data[10] = {0,1,2,3,4,5,6,7,8,9};
    struct MyClass *myObject = new_MyClass(data);

    myObject->sort(myObject);
    myObject->display(myObject);
}
```

While this is better, we still have the irritation of having to repeat the `myObject` parameter due to the fact we cannot bind a function to a `struct`.

The simplest solution to the problem is to use a pre-processor to scan the text of the program and detect functions called on structs and converting:

```
myObject.function(args) →   myObject.function(&myObject,args)
```

and:

```
myObject->function(args) → myObject->function(myObject,args)
```

This is, of course, the way C++ was first implemented as an extension of C. A text preprocessor converted the C++ extensions into pure standard C. Only later did C++ compilers become the norm.

There are many ways of adding to C to make it easier to use and in an object-oriented way, but in most cases structs and pointers to functions are all you need.

The Array At The End Of The Struct

The key thing about a `struct` is that its size has to be known so that a suitable block of memory can be allocated for its storage. What this means is that any field in a struct has to have a known size. When you allocate a string field it has to be to a fixed size string. The same holds for an array and you cannot allocate a VLA within a struct.

This is all true, but there are times when you need to create a struct with an array of a size that can vary. How to do this is obvious. Any array allocated at the end of the struct doesn't affect the layout of the struct and as such you can change its size at any time without disturbing the rest of the struct.

Originally, the accepted trick was to include a zero-length array at the end of the struct:

```
struct test {
 char c;
 int i;
 int myArray[0];
};
```

but after C99 the correct way to do this is to specify an incomplete array declaration as in:

```
struct test {
 char c;
 int i;
 int myArray[];
};
```

To make use of the array at the end you simply allocate more memory than the size of the struct which doesn't include any allowance for the array and some additional bytes for the array to use.

```
struct test *myStruct=malloc(sizeof(test) + sizeof(int[size]));
```

with this allocation you can now write:

```
myStruct->myArray[i]
```

with `i` between `0` and `size-1`

Notice that it is up to you to allocate the memory needed to extend the struct and it is up to you to make sure that the index stays within the allocated area.

If the idea of including an array at the end of a struct seems an uncommon requirement it is worth pointing out that the array is just another uncommitted area of memory and as such can be cast to whatever you like – including another struct. Using an array in this way is far more common than you might think and it is an idea worth keeping in mind.

Final Thoughts

In many ways the C struct is the key data structure in C and it is so much more than just an implementation of a data record found in other languages. It is central to the idea that C deals with bits as it is structs that let you determine what the bits mean. This is obvious to any programmer using C for the sort of tasks it was designed for – systems programming and hardware programming. Unfortunately, this is not a view shared by the programmers on the standards committees and the compiler writers, who in the main strive to make C look like a modern language and try to make it easier to optimize the code. This goes against the low-level appeal of structs and type punning in general and even makes them seem undesirable and dangerous.

For a fuller discussion, see Dive 16 which deals with the scandal of undefined behavior.

Dive 15

The Union

"Form follows function - that has been misunderstood.
Form and function should be one, joined in a spiritual union."

Frank Lloyd Wright

The union is a special type of struct that is more or less dedicated to the idea of type punning. Its very existence in the C language is a strong indication that type punning is not an accidental feature.

Union Basics

A `union` is declared in the same way as a `struct`, but memory is only allocated for the largest of the union's fields. That is, all of the fields of a union share the same memory. This sounds like a crazy thing to do until you notice that it gives you a way of working with the same bit pattern with different interpretations. You can treat the same area of memory as different types simply by using the appropriate field name. That is, using unions allows you to do reasonably well-defined type punning in C.

The syntax for a `union` is the same as for a `struct`, but you replace `struct` with `union`. For example:

```
union {
       int I;
       float F;
} myUnion;
```

This allocates four bytes – assuming `sizeof(int)` and `sizeof(float)` are both `4` – that can be used to store an `int` or a `float`. Which is stored depends on the field name used to access the `union`, `I` or `F`.

Now you can store an `int` in the `union` using:

```
myUnion.I = 42;
```

or a float using:

```
myUnion.F = 42.0;
```

In the first case, the bit pattern that represents `42` is stored in the four bytes and in the second, the bit pattern that represents `42.0` is stored in the same four bytes. Of course, there is nothing stopping you from storing an `int` and reading back a `float`, or vice versa. This is how the type punning occurs when you use a `union`.

In principle, you can pun any types, but there is a rule in C, but not C++, that it is legal to read a union member as long as it is not larger than the member most recently written. This is a perfectly reasonable restriction, but even so I can invent, admittedly unlikely, examples of reading a larger type when a smaller type has just been written, but they are all very machine-dependent. This means that, in most cases, you can use union type punning to avoid the compiler warning messages and potential undefined behavior.

As with structs and type punning, you have to take padding into account. To revisit the example in the previous chapter:

```
struct test1 {
     char c;
      int i;
};

struct test2{
    char c[4];
    char i[4];
};

union {
       struct test1 myStruct1;
       struct test2 myStruct2;
} myUnion;
```

In this case, for a 64-bit machine, `struct test1` is padded so that `int i` is on a word boundary and this means there are three padding bytes between `c` and `i`. What this means is that union of the two structs, `myUnion`, has `struct test1.c` in `struct test2.c[0]` and `struct test1.i` in `struct test2.i[0]` to `struct test2.i[3]`.

Color

For a more realistic example of a `union`, consider the way most graphics hardware stores an ARGB (Alpha, Red, Green, Blue) value in a 32-bit `unsigned int`. If you want to access each of these bytes, and also treat all four bytes as a single value, then the best way is to define a `union`:

```
union Pixel{
    struct {unsigned char b,g,r,a;};
    uint32_t value;
};
```

The only complication is that the order has to be changed to allow for the way x86 stores multiple bytes in a 32-bit word. Notice the order is machine-dependent and for Intel and Arm processors corresponds to “little endian”. You can see that the `union` is of a `struct` with four byte fields and a single 32-bit word. The four byte fields share the same memory block as the 32-bit word and on 64-bit machines no padding is needed.

Now you can declare a pixel and use it as follows:

```
union Pixel pixel;
pixel.r=1;
pixel.g=255;
pixel.b=128;
pixel.a=255;
printf("%d\n",pixel.value);
```

You can see that how you access the `union` depends on the fields that you specify. If you specify `pixel.r` then you simply work with the first byte of the memory block. If you specify `pixel.value` you work with all four bytes as a single `int`. There's no need to cast, simply use the fields that make up the `union` and you access the memory according to the type.

The type punning version of the `pixel` union is:

```
struct Pixel{
    unsigned char b,g,r,a;
};
struct Pixel pixel
pixel.r=1;
pixel.g=255;
pixel.b=128;
pixel.a=255;
printf("%d\n",(uint32_t)pixel);
```

A union can often be used in place of type punning, but it often requires a little more pre-planning to alias the fields of a suitable `union`. It is more difficult to use a union to extend structs as shown earlier because of the way a union changes the way fields are named.

Type Punning Unions

It is a little-used fact that you can cast a union to a struct or to anything else for that matter. Why not? After all, a union is just a block of memory associated with a number of different types. The block of memory is big enough to hold the largest of the types taking into account any padding that is needed. As a block of memory you can cast it to whatever you want to – but it might not make much sense.

For example:

```
union Pixel{
    struct {unsigned char b,g,r,a;};
    int value;
};
union Pixel pixel={1,255,128,255};

char (*color)[4];
color=(char *) &pixel;
```

Assuming a four-byte `int`, the memory block is four bytes in size and we can alias it to an array using an array pointer, `color`. Aliasing a `union` obeys the same rules as aliasing a `struct` and if you try this example out you will find that GCC give a warning that you are aliasing an incompatible type – the result is undefined behavior, which is a shame because it makes perfect sense, see Dive 16 for more information.

When Unions Don't Hack It

This said, unions aren't always suitable alternatives to casting. For example, if you are trying to implement an object-oriented approach to C, then aliasing a base class with its derived class is much more natural to do with a cast than a union. In fact you could argue that casting is the only way to do this job.

Consider the example given in Dive 14:

```
struct myStruct{
  int a;
  int b;
};
void myFunction(struct myStruct *myPointer){
 printf("%d\n", myPointer->a);
 printf("%d\n", myPointer->b);
}
```

If we decide to extend the struct in the future to something larger:

```
struct myStructEx{
   int a;
   int b;
   int c;
```

then we can still use the original function by casting `myStructEx` to `myStruct`, as they agree about the types of the first part of each struct:

```
myFunction((struct myStruct *) myPointer);
```

where `myPointer` is now a pointer to `myStructEx`.

Now consider how this might be implemented using a union. You would first have to create a `union` of both structs:

```
union myUnion{
       struct myStruct{
         int a;
         int b;
       };
       struct myStructEx{
         int a;
         int b;
         int c;
       };
};
```

We can now create a pointer to an instance of `myUnion`:

```
union myUnion mytest;
union myUnion *myPointer;
myPointer=&mytest;
```

How do we call:

```
void myFunction(struct myStruct *myPointer)
```

which, as originally written, expected a pointer to a `myStruct`? The only way is to cast the `union` pointer to a `myStruct` pointer:

```
myFunction((struct myStruct) *myPointer);
```

and thus we haven't avoided using a cast. The only alternative is to rewrite the function to accept a `union`, or design it this way from the start. Lack of such forethought is exactly what the cast approach aims to make up for. The cast approach is extensible in a way that the `union` approach isn't.

However, having said this, there are many approaches to this problem and this is not the end of the story if you want to invent more solutions.

Tagged Union

Although the main use of unions in low-level C programming is to perform type punning, this is not what more general programmers tend to think a union is for. Many programmers, and especially those more familiar with higher-level languages, see `unions` as ways of saving storage or creating flexible data structures often called variants or tagged unions.

For example, suppose you have a name record which sometimes has a telephone number as a string and sometimes as an integer. You could store this as:

```
struct {
        int type;
        union {
            char numstring[10];
            int numint;
        } phone;
} person;
```

Notice that the `phone` fields are a `union` of `char[10]` and `int`. The `type` field is the tag indicating which type is to be used. For example:

```
person.type = 1;
strcpy(person.phone.numstring,"1234");
person.type = 0;
person.phone.numint = 1234;
```

When trying to access the fields you would test the `type` field first.

In this example, it is obvious that you should parse the variations on the telephone field into a standard format, but there are situations where this doesn't make sense.

If the compiler supports anonymous unions we can get rid of the phone field which seems unnecessary:

```
struct {
        int type;

        union {
            char numstring[10];
            int numint;
        };
} person;
```

Now we can write:

```
person.type = 1;
strcpy(person.numstring, "1234");
person.type = 0;
person.numint = 1234;
```

GCC supports this use of an anonymous union.

Final Thoughts

Unions are the preferred way of type punning mainly because they have to be explicitly declared. If you want to alias two types then you have to create a union that implements this. If you use casting then you don't have to prepare in advance, you can simply cast as needed. This makes it harder for the compiler to detect when you use an alias.

Dive 16

Undefined Behavior

"The fact is, undefined compiler behavior is never a good idea. Not for serious projects."

Linus Torvalds

Undefined behavior is something that every C programmer should be aware of and take steps to protect themselves against. It all stems from the way optimization has become something of a game for compiler writers. The real problem actually started well before undefined behavior emerged to make it worse.

An Optimization Too Far

You might think that the highest consideration of a compiler writer is to compile a program in such a way that does not alter the meaning of a program. Sadly, this doesn't seem to be the case any longer, the goal of optimization has now pushed compilers into the realms of inventing their own idea of what a program should do.

For example, in many low-level programs there is a need to waste some time, which obviously conflicts with the goals of optimization as they are to save time not waste it. To produce a rough time delay, often the simplest thing to do introduce a no-op loop:

```
for(int i=0; i<1000; i++){};
```

This is often needed if you are interacting with the hardware and need to pause while the hardware catches up. In theory this should never be necessary as there should always be a status bit that you can test to see if the hardware is ready but in practice this isn't always the case. As a result a typical program might have multiple no-op loops which, at the time of writing, are all automatically removed by GCC, even using its lowest level of optimization.

Is this reasonable?

I would argue that it isn't, as no sane programmer would write a no-op loop of this sort by accident – it cannot be a mistake. In addition, its only purpose can be to waste time and for a compiler to eliminate it is working against the intention of the programmer.

There are two solutions to this problem – switch off optimization or mark the loop variable as `volatile`:

```
for(volatile int i=0; i<1000; i++){};
```

This tells the compiler that the loop variable is special in that it could be modified by "external" influences. For example, you might mark a variable as volatile if it corresponds to a hardware register that is updated independently of the program – a timer or an input device.

The `volatile` modifier has the desired effect and the compiler no longer removes the loop, but it has additional effects. The most important is that `volatile` stops the compiler from moving the variable into a register. A standard optimization is to move the loop index from a location in memory, into a register. The register is used for the duration of the loop and the final value is moved back into memory when the loop ends if the variable is still in scope. A more important issue, however, is that marking the index variable as `volatile` gives the wrong impression of the intent, unless other programmers also know that this is a "trick" to stop the compiler removing no-op loops.

Removing no-op loops is an example of optimization by dead-code elimination. The problem is that one programmer's dead-code is another's essential behavior. In the standards, dead-code is loosely defined as code that has no observable effects, i.e. an operation that seems to do nothing and has no side effects.

The problem is that it is very difficult to work out what has no observable effects. Assigning to a variable that you then don't use has no observable effects as long as it isn't a hardware-based register controlling something. Similarly a no-op loop has no observable effects unless you include the time it takes to run and time is not on the list of considerations of the standards writers. This clearly marks them out as having a very particular set of applications in mind when constructing a standard for C and its isn't one that fits particularly well with C being used in low-level, hardware-oriented, code.

For many C programs time is important and should not be optimized away to nothing.

Dead code removal can also have effects when the compiler just can't anticipate the side effects of human intervention. Clearly any obvious interaction with the outside world is recognized as "observable" and it isn't ever treated as dead code. However, when interrupts and traps come into the picture things aren't so obvious. For example, many microprocessor systems

have additional controls via a JTAG probe. In this case sometimes it is a good idea to halt the program at a position of interest using an infinite loop:

```
while(1);
```

the rest of the program follows and the idea is that the JTAG is used as a hardware debugger to restart and trace the program. Of course, what happens is that the optimizing compiler removes all of the code following the `while` as it is clearly unreachable and hence clearly dead code.

Another example, is when an exception is an observable effect, but the dead code is still eliminated. For example:

```
int a;
int b = 0;
a/b;
```

should throw an exception, but on GCC in default optimization mode it doesn't. The code is clearly dead as the result of `a/b` is never used, but it does have a side effect in that division by zero should always throw an exception. If you change the final line to:

```
int c = a/b;
```

then the exception occurs even though `c` is never used! You could say such a stupid program doesn't deserve consideration, but if you were writing a test to see if an exception was thrown you might want to disagree. Even stupid programs sometimes have their uses. Interestingly, marking `a` and `b` as volatile changes things, but only in that `a` and `b` are not optimized away, only the operation of `a/b` is removed:

```
volatile int a;
volatile int b = 0;
a/b;
```

There are many other examples, but the point here isn't so much to complain about compilers and standards which go against what you might want to do, but to expose the fact that compiler optimization isn't a simple concept or process. In practice, there are generally well-known ways around these problems – either turning off the offending optimization or making sure that the code has observable effects. This is not to say that programmers don't waste a great deal of time because of unwanted optimizations – far more time than that due to compiler bugs and errors. For example if you haven't seen it before trying to work out why a timing no op loop appears to take no time at all can take a great deal of effort.

It is also important to realize that optimizing compilers are a good thing, despite the occasional clash with the programmer's intent. It is very rare that I have been able to improve on the assembly language produced by GCC and reading it is often an education in itself.

Undefined Behavior

The current concern is undefined behavior, UB – what it is, what it means and how to avoid it. However, as we have just seen, before undefined behavior hit the C headlines, we had unobservable behavior to cope with. In many ways the two are very similar, but undefined behavior takes on a more aggressive role.

Undefined behavior is a term used to specify that a particular operation has no meaning or is unpredictable. The idea is that the language specification refuses to specify what is to happen, leaving it up to someone else to supply an interpretation. This seems innocent enough, but over time it has morphed into something of a monster.

There is a great deal of argument about when the notion of undefined behavior first arose and much of it is about the placement and interpretation of commas in standards. Standards are not always particularly clear. What is clear is that, initially at least, undefined behavior was used to simply mean that the standard did not cover the situation. In this sense, if you wrote code that included undefined behavior then your program was not covered by the standard.

Perhaps the high point of undefined behavior's travel from reasonable to unreasonable was the famous "nasal demons" post in 1992 on a fairly "serious" newsgroup, comp.std.c:

" Undefined behavior -- behavior, upon use of a nonportable or erroneous program construct, ... for which the standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose."*

If you think that this humorous comment is extreme, a few posts earlier we have:

"At this point, the compiler is at liberty to generate a program which formats your hard disk drive. In fact, it probably should."

The idea that if a program contained undefined behavior then all bets were off grew and continues to grow from this point. It is the programmer's equivalent of the principle of explosion. If you have two logical premises, P and !P, and you assume that they are both true, then anything can be proved true. The reasoning is:

P is true so P or Q is true no matter what Q is. But as !P is true P or Q implies that Q is true. Hence any statement can be proved true if P and !P are assumed.

So rejecting the law that only one of P or !P can be true, leads to a situation in which logical demons can fly out of your nose.

Of course, in practice, the existence of the principle of explosion is an argument for anything that invokes it is meaningless. So too with undefined behavior – its exploitation produces meaningless program semantics.

And yet we still have the confusing idea that undefined behavior is at the same time somehow useful, and yet something that programmers need to identify and eliminate if they want to avoid nasal demons.

It also gives you some idea of how out of touch compiler writers can be when they don't appreciate the uses of the language they are trying to compile.

There is a huge difference between stating that the standard does not define what happens in a particular case and thinking that this gives you carte blanche to do whatever you want. It seems that what happened was that the standards committee discovered it was easier to leave some behavior to the machine the program was running on to deal with, but instead of marking it as machine-dependent or implementation-dependent they chose to represent it as simply being outside of the standard and then the rest followed.

UB in Practice

The reputation of undefined behavior, UB, has caused many programmers to focus on it and avoid anything that has been defined as such. This is an overreaction because many undefined behaviors are perfectly well-defined in practice and, in many cases, desirable rather than avoidable. The important thing to realize is that despite being threatened with nasal demons and reformatting of disks, this rarely happens.

In practice undefined behavior is mostly a nuisance just like the optimizations that don't take account of what might be observable. Indeed one of the examples given of dead code deletion could just as well be listed as undefined behavior as dividing by zero is listed as such. So when you write:

```
int a;
int b = 0;
int c = a/b;
```

you have already got undefined behavior, even if you then don't make use of `c` and have deadcode. In practice, what happens is not nasal demons but something much worse – an exception.

Of course, division by zero is something you need to protect against – even if it wasn't undefined behavior. The usual way is to do this is to use an `if` statement:

```
if(b!==0) c = a/b;
```

or make use of lazy evaluation:

```
c = b & a/b;
```

Most other undefined behaviors are treated in a similar way and as long as you write a good program that protects itself against the obvious, nasal demons will ignore you. For example, undefined behavior that falls into this category includes:

- Accessing array outside of bounds
- Modifying a string literal
- Dereferencing/accessing a null pointer
- Changing a variable twice between sequence points
- Shifting by negative amounts
- Accessing uninitialized data

These and many others are very reasonable and easy-to-deal with, but some are more problematic, signed overflow and strict aliasing being the most important.

Divide By Zero and Undefined Behavior

It is worth explaining the crazy position of divide by zero in C99:

> *C99 6.5.5p5 - The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.*

If you need a clear demonstration that undefined behavior is completely mad and unacceptable, this is it.

You will find some programmers arguing that `x/0` is infinity because `x/y` with `x` and `y` positive gets bigger as `y` tends to `0`. This is true, but mathematicians define `x/y` as undefined if `y` equals `0`. The reason is that the operation makes no logical sense. For example: `x.0 = 0` doesn't have a unique solution and `0` is the only number that doesn't have a multiplicative inverse.

So it looks as if deciding that division by zero is undefined is justified – but that would be wrong. The idea that encountering `y/0` is a permission to do anything the compiler writer can think of is nonsense. A program that includes `y/0` may be an incorrect program, but the correct behavior is to indicate that the operation is illegal. The program has to report the error or throw an exception or at the most general perform some action which is machine dependent.

Division by zero is a runtime error not undefined behavior.

Signed Overflow

Even though the C23 standard defines signed integers as using two's complement, it still defines signed overflow as undefined, even though it is fully defined by any reasonable interpretation of machine arithmetic using two's complement.

As already explained in Dive 1, signed overflow is well-defined in terms of bits, but not so much in terms of what it means. Signed overflow results in going from the largest positive value to the largest negative value by adding one. It breaks the laws of arithmetic as this is an example of where `a + 1 <a` , but is this enough to declare that it is undefined behavior?

Treating signed overflow as undefined behavior is also an example that demonstrates the reason why making it undefined provides opportunities for optimization that otherwise would be difficult.

Consider the following:

```
int a = INT_MAX;
      if ((a + 1) < a)
        printf("Integer overflow!");
```

This simply adds one to the maximum positive integer and hence forces an overflow, but you will never see the error message printed. The simple reason is that, as signed overflow is undefined behavior, no program conforming to the standard can exhibit it and so, by the laws of arithmetic, `a + 1` is always bigger than `a` and so the whole if statement can be removed as dead code! This undoubtedly would make the program run faster, but if the `if` statement was intended to actually test for an overflow then it would also make the program incorrect. For example:

```
      if ((a + 1) < a)
        printf("Integer overflow!");
      else
        printf("%d",a+1);
```

This program never catches the overflow error because the if statement is modified so that the else clause is always obeyed. Of course, "nasal demons" means that anything could happen, but in practice the undefined behavior isn't used to punish the programmer – it is used to "optimize" the program.

Notice that this optimization causes the very error condition, i.e. signed overflow, that the undefined behavior and the program was supposed to prevent. In this case "nasal demons" would be a better choice than simply deleting the test for overflow as it might alert you to the condition.

So how do you detect signed overflow?

The whole problem is that you have to detect it without causing it and this isn't simple for a general signed integer expression.

For example, how do you check that:

```
b*c + a*d
```

doesn't cause a rollover before you compute it? It is very difficult because you have to check that each stage in the calculation doesn't cause an overflow.

There are also compiler extensions to make such tests easier. For example, GCC has the `__builtin_`*`operation`*`_overflow` functions for operations of `add`, `sub` and `mul` for different integer types. The function avoids signed overflow by performing infinite precision arithmetic and then checks to see if the result can be stored in the third parameter without loss of bits. If it can't, overflow would occur if the calculation was done in the specified type and the function returns `true`. Using these functions is not easy and they are compiler-specific.

You can also ask GCC to generate a `SIGABRT` signal when signed overflow occurs by using the `-ftrapv` option:

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

void signalHandler(int sig) {
 printf("Overflow detected\n");
}

int main(int argc, char **argv)
{
 signal(SIGABRT, &signalHandler);
 int a=INT_MAX;
 int b=a+1;
```

As long as this is compiled with the `-ftrapv` option on GCC, then the overflow will be detected and you will see the overflow message.

Another solution is to avoid the problems of signed arithmetic and rollover by performing the exactly identical unsigned arithmetic. Remember that the only difference between signed and unsigned is how you interpret the bits. That is, if you have a set of signed variables and you cast them all to unsigned, perform the calculation and cast the result to signed, you will have the same result as if you had performed the signed calculation. The only scope for a problem here is when you cast back to signed the compiler could decide that assigning a positive value larger than `INT_MAX` to a signed variable is overflow – no reasonable compiler does this.

For example:

```
int a = INT_MAX;
      if( (signed)((unsigned)a+(unsigned)1) < a)
        printf("Integer overflow!");
      else
        printf("%d",a+1);
```

In this case the arithmetic is performed unsigned and overflow is fully defined. Converting back to `signed` reveals that there has been an overflow and we see the appropriate message, even with optimization.

In this case, it seems clear that the optimization obtained by making signed overflow undefined behavior isn't worth the trouble it causes.

Strict Aliasing

This is the undefined behavior which causes the most trouble of all because it goes against the very fundamentals of what C is designed to be good at. As in the case of signed overflow, it is another example of undefined behavior in the service of a fairly unwanted optimization.

Strict aliasing simply means that if two pointers are of incompatible types then they cannot be referencing the same data. Unfortunately, the rules as to which types are compatible are difficult to interpret, but roughly they come down to compatible types are any type that differ only by a qualifier or signed/unsigned and a char type is compatible with everything. Structs are only compatible if all of their fields are compatible.

This all sounds very abstract – let's take a look at how this works in practice with a simple example. Consider the function:

```
int alias(int *a, float *b)
{
  *a = 0;
  *b = 1.0;
  return *a;
}
```

In this case the two pointers are to incompatible types and so the compiler can assume that they do not reference the same thing. As they do not reference the same data, the function can be optimized to:

```
int alias(int *a, float *b)
{
 *a = 0;
 *b = 1.0;
 return 0;
}
```

Now consider what happens if we break the strict aliasing rule:

```
int main(int argc, char **argv)
{
  int a = 42;
  printf("%d\n", alias(&a, (float *)&a));
  printf("%d\n", a);
}
```

You can see that we are passing an `int` pointer and a `float` pointer to the function and hence `a` and `b` in the function do reference the same data. But the fact that the compiler thinks this doesn't happen results in the function returning `0` even though `a` in the main program should be set to 1.0 by the function call.

There are lots of examples where the assumption that pointers of two incompatible types cannot refer to the same data, produce very similar results. You might think that aliasing `int` to a `float` isn't useful and perhaps even outright wrong, but you get the same effect with `int` and `long` which are also not compatible:

```
long alias(long *a, int *b)
{
  *a = 0;
  *b = 1;
  return *a;
}
int main(int argc, char **argv)
{
  long a = 42;
  printf("%d\n", alias(&a, (int *)&a));
  printf("%d\n", a);
}
```

This should display `1` and `1` but with optimization it displays `0` and `1` when compiled with GCC. This strange result is because as it is assumed that `*a` and `*b` reference different things the `return *a` can be converted into

```
return 0;
```

Both assignments are carried out and hence `a` in the main program is `1`. This is all the more surprising when you learn that on most modern systems `int` and `long` are both four bytes and therefore highly compatible in general terms.

The worst case is that two structs that have incompatible fields are incompatible. For example:

```
typedef struct {int x;} struct1;
typedef struct {short y;short z;} struct2;

long alias(struct1 *a, struct2 *b)
{
   a->x=0;
   b->y=1;
   return a->x;
}

int main(int argc, char **argv)
{
  struct1 a;
  a.x=42;
  printf("%d\n", alias(&a, (struct2 *)&a));
  printf("%d\n", a.x);
}
```

In this case we have two structs which are compatible from the point of view of padding, endianism and total size – but not by the type of fields. Put simply `struct2` treats the `int` in `struct1` as two shorts. If you try this out you will discover that once again the function has been optimized and the function returns `0` and `1` because it is assumed that `a` and `b` reference different things and so the `return a->x` can be optimized to `return 0`. Notice that setting `b->y` to `1` also sets `a->x` to `1`.

What is to be done about these erroneous optimizations?

One very prevalent attitude is to say that the programmer should avoid undefined behavior and not doing so simply results in the construction of an invalid program. This is to ignore the usefulness of the idea of type punning, especially for structs. You could avoid some of these problems by using suitable unions in place of type punning, but sometimes this just isn't appropriate. Some even suggest that the correct way to do the job is to use `memcopy` to create a bit copy of the original datatype to another area of memory and treat that like the alternative type in the hope that the compiler notices the idiom and optimizes it away! This just reveals how ludicrous the situation of defining something, that makes C the language it is, as undefined behavior.

The best solution, if using GCC, is to always compile your programs with the `-fno-strict-aliasing` option. If you try this out with all of the previous examples, you will discover that they work as the programmer intended rather than as the compiler writer fantasized.

The last word should go to Linus Torvalds:

> *"The idiotic C alias rules aren't even worth discussing. They were a mistake. The kernel doesn't use some "C dialect pretty far from standard C". Yeah, let's just say that the original C designers were better at their job than a gaggle of standards people who were making bad crap up to make some Fortran-style programs go faster. They don't speed up normal code either, they just introduce undefined behavior in a lot of code. And deleting NULL pointer checks because somebody made a mistake, and then turning that small mistake into a real and exploitable security hole? Not so smart either."*

Dive 17

Exceptions and the Long Jump

"The young man knows the rules, but the old man knows the exceptions."
Oliver Wendell Holmes, Sr.

Exceptions are a great idea, but they are generally associated with higher-level languages than C. As a result many a C programmer thinks that C doesn't support exceptions, even if they are familiar with the instructions that can be used to implement an exception-like mechanism. First let's find out what an exception is and why it is different from most other flow of control constructs. Then let's see how C can do the job.

The Idea of an Exception

A great example of what exception handling is all about can be found in nearly all modern languages. For example, Java, JavaScript and C# has try-catch and Python has try-except, which works in much the same way.

In many languages you can write something like:

```
try:
  block of code
catch:
  error handling code
```

If any instructions in the block of code cannot be completed for any reason, an exception is raised and the catch clause is activated. Most modern languages have a similar construct, but perhaps with the syntax changed slightly.

This seems nothing more sophisticated than:

```
if error then handle error
```

but it is much more. The block of code may call functions and if an error occurs in one of the functions we have the problem of what to do. A typical C approach is to terminate the program and return an error code. This isn't particularly user-friendly and high-level languages implement exceptions to provide a way to handle errors.

Of course, the obvious thing to do is detect the error and attempt the operation that caused it again with modifications in the hope that things work this time around. Sometimes this is easy, but usually it isn't due to the need to get back to the code that started the process that led up to the error.

For example, consider a simple C function:

```
int divide(int a, int b){
   return a/b;
}
```

If this is called with arguments that cause an error:

```
int result = divide(1,0);
```

it will result in a divide by zero error in the `divide` function. The error occurs in the function, but there is no way that the function can handle the error – it has just been given `a` and `b` with no way to know what to do if `b` is `0`. To handle the error, the code that called the function has to call it again, but with sensible arguments that don't cause an error.

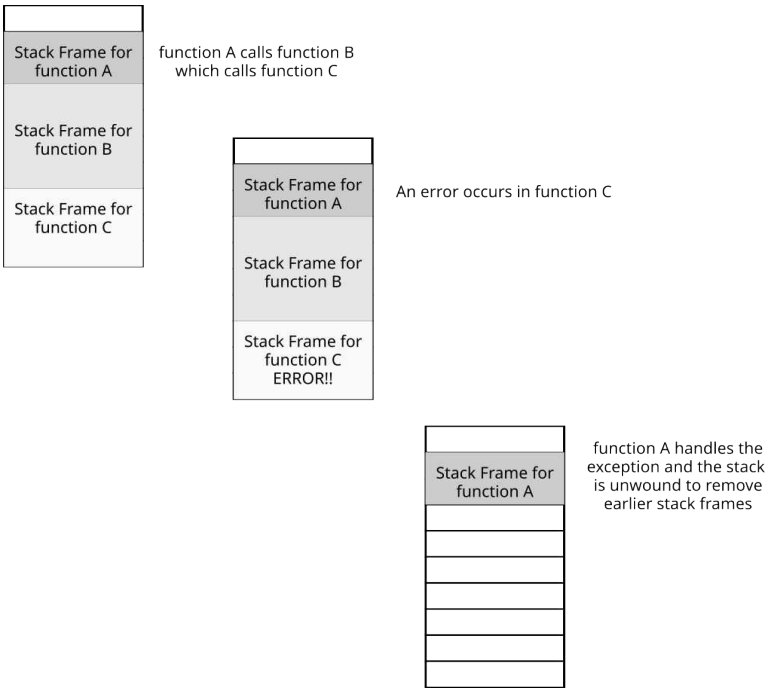
The general point is that an error will typically occur within a function which is in no position to fix the error because the cause of the error is in code which called the function, or the code which called a function which then called the function and so on...

Even when an error can be handled locally, it may still be better to pass the problem back to a much earlier calling function. For example, if a function tries to download a file over the network and it times out, then the function doing the downloading could just retry the download. However, if the timeout function passes the error back up to a calling function, then it may have the logic to decide to try to download a different file. In other words, where an error is best handled in the call chain, is something that depends on the task and the type of error.

The problem is that we need to return control to an earlier calling program while wiping out anything that the functions that have been called may have done. We need to return to the point in the code where the error is to be handled in a state as if the action that resulted in the error had never been tried.

This is what we mean by an "exception" - a piece of code that attempts to do something, usually by calling a function and if an error occurs, perhaps deep in a sequence of calls, then an exception is raised and control passes back to the code that is to handle the exception so that it can try again. The state of the system has to be restored as much as possible to what it was before the functions were called and the error occurred.

The easiest and most logical way to implement the state change needed to implement an exception is to “unwind” the call stack back to the point where the code that is handling the error started the function calls.



As all of the function’s local variables are stored on the stack, this essentially obliterates all trace of the functions being called. Well not quite, because adjusting the call stack cannot do anything about any global side effects that the functions caused. For example, if the function printed something, opened a file or modified a global variable, then these things are not automatically undone by the exception. In the case of having printed something then it might not be possible to undo what has been done, but usually the `catch` portion of the exception handler can perform the final cleanup by adjusting global variables and closing files etc.

Once everything is cleaned up as much as possible the code that is handling the error can have another go at the operation. Of course if this causes an exception then again we need to do something different and eventually we are going to have to give up and report an error. Ideally the code should ask for human assistance in working out what to do next but most code still just posts an error number and halts.

This outline gives you the basics of exceptions and exception handling. Modern languages usually allow the catch clause to define which errors they will handle and if they can't handle the particular error the exception is passed up to the next function to see if it can handle it and so on. Code can also raise or throw an exception any time it is needed, generally using a command like:

```
raise exception type
```

Notice that an exception breaks the usual flow of control within a program. It is almost as if there were two flow of control graphs – one giving the path through the program when things go right and one giving an alternative path when things go wrong. Exceptions may break the usual flow of control, but they do it in a very specific way by slowly moving back up the function call stack. Mechanisms that work with the natural call and return of function are usually called "structured" as opposed to "unstructured", which implies that anything can happen. So you could say that a `goto` is unstructured and exceptions are structured.

Setjmp Non-local Jumps

The key to implementing an exception-like error handling mechanism in C is the non-local jump. This is like a `goto` on steroids, but it lends itself to structured use. The `setjmp` function stores the current state of the program in a buffer and returns a `0`. The program can then carry on as normal until a `longjmp` function call occurs with the same buffer used in `setjmp`. When this happens, the stored environment is restored and control is transferred back to the `setjmp` function, but this time it returns the value of the second `longjmp` parameter so that you can use it to tell that this is a restart.

That is:

```
setjmp(buffer);
```

stores the state in `buffer`, a `jmp_buf`, and returns `0`.

After this the program carries on as normal until it meets:

```
longjmp(buffer, n);
```

when the environment is restored and control is transferred back to the `setjmp`, which returns *n*. Notice that the restored environment wipes out any function calls, i.e. it unwinds the stack back to its state when the `setjmp` was first called. Notice that the `jmp_buf` is an array and its size depends on the system – it has to be large enough to store the status.

Using `setjmp` and `longjmp` is fairly easy:

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

static jmp_buf jumpBuffer;

int division(int a,int b){
    if(b==0){
        longjmp(jumpBuffer,1);
    }else{
        return a/b;
    }
}

int main(int argc, char *argv[])
{
    if(0==setjmp(jumpBuffer)){
        int result=division(1,0);
    }else{
        printf("divide by zero");
    }
    return 0;
}
```

You can see that the `division` function tests for the divide by zero error and performs a `longjmp` if it happens. The main program simply tests the return value of `setjmp`. The first time it is called it returns `0` and hence the `if` clause calls division. This in turn triggers the `longjmp` with the result that control returns to the `setjmp` call which this time returns `1` and so the `else` clause is called. You can think of the `longjmp` as a "raise the exception" instruction and the `if` statement as a try-catch rather than if-else.

Try Catch Macros

You can make this correspondence explicit by defining some simple macros:

```
#define InitalizeException static jmp_buf jumpBuffer
#define raise(E)  longjmp(jumpBuffer,E)
#define try if(0==setjmp(jumpBuffer))
#define catch else
```

With these definitions the original example can be written:

```
InitalizeException;

int division(int a,int b){
    if(b==0){
        raise(1);
    }else{
        return a/b;
    }
}

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

    try{
        int result=division(1,0);
    }
    catch{
        printf("divide by zero");
    }
    return 0;
}
```

Of course, this is just “syntactic sugar”, but it is all that other languages provide. It lacks some more sophisticated features. For example, it doesn’t let you test for the exception type and it doesn’t allow for an exception to be re-raised if the try-catch can’t handle it.

Handle or Reraise

It can be difficult to keep track of all of the `setjmp` instructions in a program. As an alternative we can extend the macros given above so that we can set multiple try-catch constructs and expect them to be handled correctly. The only real cost of this exercise is that now the `catch` clause has to signal that it has either handled the exception or re-raised the exception.

The idea is to maintain an array of `jmp_bufs` and a current index which indicates the very latest state as the result of a `setjmp`. Now when a `try` occurs, the index is incremented and the state of the system is stored in the array. When an exception is raised, the `longjmp` uses the current array element as its buffer. If the catch clause handles the exception, the index has to be decremented. If the catch clause re-raises the exception the index is decremented and a `longjmp` performed, using it to take us to the previous try catch. To get things started we also need an initialization function which sets up a top-level try-catch that aborts the program if no other try-catch handles the exception. We also need a global `ErrorNo` variable to store the current error number code.

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

#define try        \
    ExceptDepth++; \
    ErrorNo=setjmp(jumpBuffer[ExceptDepth]);\
    if (0 ==ErrorNo )

#define catch \
    else

static jmp_buf jumpBuffer[10];
static int ExceptDepth = -1;
static int ErrorNo = 0;

void InitalizeException(void)
{
    ExceptDepth++;
    ErrorNo = setjmp(jumpBuffer[ExceptDepth]);
    if (0 == ErrorNo)
    {
        return;
    }
    else
    {
        printf("unhandled error %d\n", ErrorNo);
        exit(ErrorNo);
    };
}

static void raise(int E)
{
    longjmp(jumpBuffer[ExceptDepth], E);
}

static void reraise(int E)
{
    ExceptDepth--;
    longjmp(jumpBuffer[ExceptDepth], E);
}

static void handled(){
    ExceptDepth--;
}
```

```
int division(int a, int b)
{
    if (b == 0)
    {
        raise(-1);
        return 0;
    }
    else
    {
        return a / b;
    }
}

int main(int argc, char *argv[])
{
    InitalizeException();

    try
    {
        int result = division(1, 0);
    }
    catch
    {
       //reraise(ErrorNo);
        handled();
        printf("divide by zero\n");
        printf("depth %d\n", ExceptDepth);
        printf("ErrorNo %d\n", ErrorNo);
    }
    return 0;
}
```

The example `main` program handles the expression and removes the entry in the array. If you choose to re-raise the exception then the previous `jmpbuf` element is used and in this case it takes the control back to `InitalizeException` and so ends the program. If there was another try-catch in a function next up in the call chain then its catch clause would be executed with the same error number.

Is this approach a good idea? Probably not. Keeping an array of exceptions to return to is likely to be a fragile construct. It works, but what about it being used in ways you never thought of. It is difficult to make this sort of construct robust. This doesn't mean that the `setjmp` and `longjmp` are a waste of time, but they are probably best used where they do direct and obvious good and without any syntactic sugar.

Restrictions

There are some subtle points about using `longjmp`. The first is that on return to a `setjmp` the stack is restored to its state at the `setjmp`. What this means is that any changes to local variables after the `setjmp` are lost. This is perfectly reasonable and what you would expect. However, it is possible that a compiler could optimize the code so that local variables are placed in registers. In this case, the restore of the stack might not be exact. As a result, in principle, if not so much in practice, local variables modified after the call to `setjmp` and before the call to `longjmp` should be regarded as undefined. In principle you can make them defined by using the `volatile` keyword in their definition.

A more important restriction in practice is that, because `setjmp`/`longjmp` are implemented in assembler, `setjmp` should only be used in a small number of circumstances:

- As the test expression of a selection or iteration statement (such as `if`, `switch`, or `while`).
- As one operand of an equality or comparison operator that appears as the test expression of a selection or iteration statement. The other operand must be an integer constant expression.
- As the operand of a unary '!' operator that appears as the test expression of a selection or iteration statement.
- By itself as an expression statement.

This means that

```
state = setjmp(jumpBuffer);
```

is safe but something like:

```
state = setjmp(jumpBuffer[i++])+base;
```

is not.

Finally, it is important that the function that the current `setjmp` is in hasn't returned before the `longjmp` is executed. The reason should be obvious – the stack cannot be restored to its state in the function if control is no longer in the function. As the documentation says:

> *"If you longjmp to a return point that was established in a function that has already returned, unpredictable and disastrous things are likely to happen."*

This seems very reasonable.

Linux Signals

So far we have used `longjmp` to deal with errors that we can detect using code. Some errors can't easily be checked out in this way and the operating system deals with them. Ideally we would like to convert an operating system-handled error into an exception-handled error. This complicates things because you have to use a different method for Linux and Windows.

Linux is perhaps the easier to work with as it implements signals in a reasonably consistent way as defined by the POSIX standard. If you need more information on how POSIX implements software interrupts using signals, see *Applying C For The IOT With Linux,* ISBN: 978-1871962611. Windows also implements signals, but only to be compatible with POSIX.

There is one more complication. The standards do not specify if the signal state should be stored as part of the environment. Under some operating systems it is, and under others it isn't. The function:

```
sigsetjmp(buffer,0);
```

will save the environment without the signal state. If the second parameter is non-zero then it will save the signal state.

If you use `sigsetjmp` then you have to use:

```
siglongjmp(buffer,value);
```

to restart.

Consider how to allow division functions to raise a signal. If you use:

```
int myDiv(int a, int b) {
    return a / b;
}
```

then, as the program stands, it would simply perform the default action and terminate without any signal.

If we define a handler for `SIGFPE` then we can use the `siglongjmp` to return to the program and continue:

```
void signalHandler(int sig)
{
    printf("SIGFPE\n");
    siglongjmp(jumpBuffer, 1);
}
```

Setting up the signal handler is the same as always and the complete program is:

```
#define _POSIX_C_SOURCE 200809L
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <setjmp.h>


static jmp_buf jumpBuffer;

int division(int a, int b)
{
    return a / b;
}

void signalHandler(int sig)
{
    printf("SIGFPE\n");
    siglongjmp(jumpBuffer, 1);
}

int main(int argc, char **argv)
{
    sigset_t mask;
    sigfillset(&mask);
    struct sigaction psa = {0};
    psa.sa_handler = signalHandler;
    psa.sa_mask = mask;
    sigaction(SIGFPE, &psa, NULL);

    if (0 == sigsetjmp(jumpBuffer, 0))
    {
        printf("answer %d\n", division(6, 2));
        printf("answer %d\n", division(6, 0));
    }
    else
    {
        printf("can't do it\n");
    }
    printf("program continues\n");
    return (EXIT_SUCCESS);
}
```

The `SIGFPE` signal is raised for any arithmetic error. If you want to see this program working then you have to run it without a debugger active. The reason is that the debugger will generally automatically intercept any fatal signals before the program has time to handle them. Try running it from the command line.

goto versus longjmp

C has a `goto` instruction and at first it seems similar to the `longjmp` function, but it isn't. The C `goto` can only be used to transfer control within a function to a label applied within that function. The `longjmp` can be used to transfer control between functions, which could even be in different source files. The `goto` should never be needed as modern C has enough control structures to make an uncontrolled jump to an arbitrary location unnecessary and, if used without a really good reason, it is confusing and obscure. There are occasions when a `goto` can be considered a reasonable choice and these are exactly the occasions when the `longjmp` is useful. If there is some exceptional occurrence, usually an error, then a `goto` might be acceptable as a way of continuing the flow of control, but even here there are usually better ways.

The `longjmp` can also be a source of confusion. The fact that the target of the `longjmp` can be set multiple times means that if you use:

`longjmp(`*`jumpBuffer`*`)`

then where you end up depends on what is stored in *`jumpBuffer`*. One solution is to always use a different *`jumpBuffer`* so that they act like labels in a `goto`. For example:

```
longjmp(startComputation)
longjmp(getUserInput)
```

where each *`jumpBuffer`* is set at the corresponding location. This seems like a very good approach to controlling `longjmp`.

To summarize:

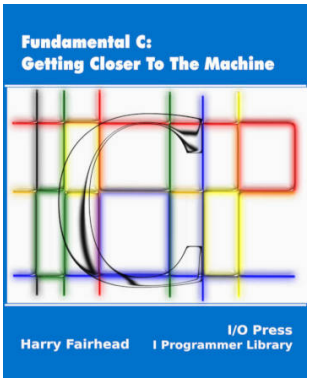
- `goto` is rarely needed and always confusing
- `longjmp` is sometimes needed, but still potentially confusing

Final Word

C doesn't have an exception mechanism, but `longjmp` provides a reasonable way to handle runtime errors. It isn't perfect due to the difficulties of defining what should be stored by a `setjmp` and what should be restored by a `longjmp`. This makes using `longjmp` hardware-dependent. Even in an ideal world, using `longjmp` is potentially confusing. It is best used sparingly and reserved for when it solves a real problem, rather than trying to implement a general exception-handling mechanism that operates throughout the program.

It is also worth reminding ourselves that one programmer's exception is another programmer's conditional. You can generally let an exception happen or you can write code that detects it before it happens.

Index



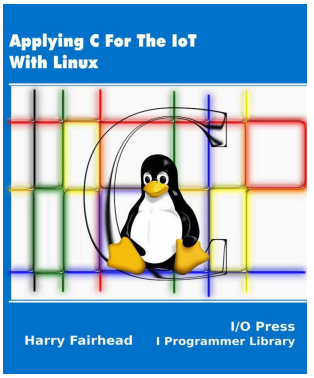
Fundamental C: Getting Closer To The Machine

ISBN: 978-1871962604

If you want to get your knowledge of C up to speed then this is the book for you. This book explores C from the point of view of the low-level programmer and keeps close to the hardware. It covers addresses, pointers, and how things are represented using binary and emphasizes the important idea is that everything is a bit pattern and what it means can change.

For beginners, the book covers installing an IDE and GCC before writing a Hello World program and then presents the fundamental building blocks of any program - variables, assignment and expressions, flow of control using conditionals and loops.

When programming in C you need to think about the way data is represented, and this book emphasizes the idea of modifying how a bit pattern is treated using type punning and unions and tackles the topic of undefined behavior, which is ignored in many books on C.

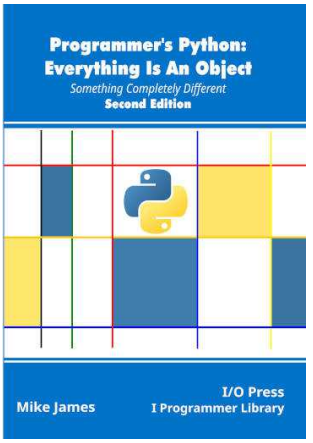


Applying C For The IoT With Linux

ISBN: 978-1871962611

If you are using C to write low-level code using small Single Board Computers (SBCs) that run Linux, or if you do any coding in C that interacts with the hardware, this book brings together low-level, hardware-oriented and often hardware-specific information.

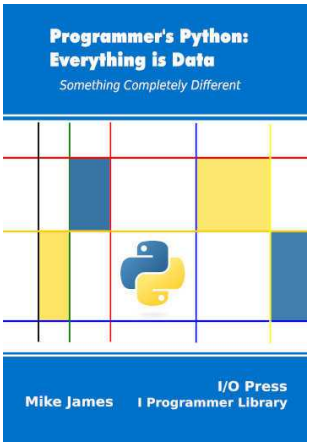
It starts by looking at how programs work with user-mode Linux. When working with hardware, arithmetic cannot be ignored, so separate chapters are devoted to integer, fixed-point and floating-point arithmetic. It goes on to the pseudo file system, memory-mapped files and sockets as a general-purpose way of communicating over networks and similar infrastructure. It continues by looking at multitasking, locking, using mutex and condition variables, and scheduling. It rounds out with a short look at how to mix assembler with C.



Programmer's Python: Everything is an Object, Second Edition

ISBN: 978-1871962741

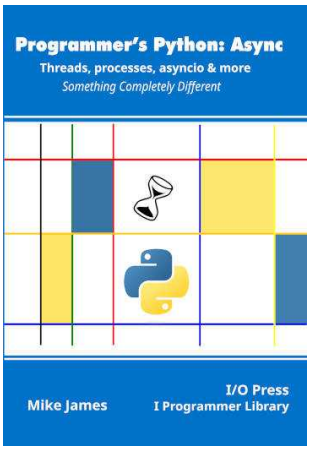
This is the first in the Something Completely Different series of books that look at what makes Python special and sets it apart from other programming languages. It explains the deeper logic in the approach that Python 3 takes to classes and objects. The subject is roughly speaking everything to do with the way Python implements objects - metaclass; class; object; attribute; and all of the other facilities such as functions, methods and the many "magic methods" that Python uses to make it all work.



Programmer's Python: Everything is Data

ISBN: 978-1871962595

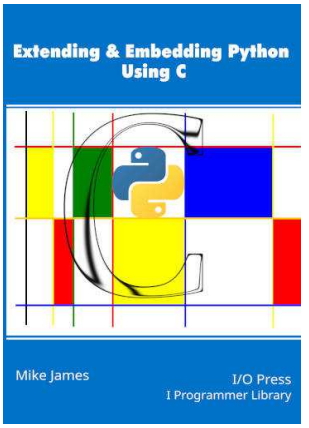
Following the same philosophy, this book shows how Python treats data in a distinctly Pythonic way. Python's data objects are both very usable and very extensible. From the unlimited precision integers, referred to as bignums, through the choice of a list to play the role of the array, to the availability of the dictionary as a built-in data type, This book is what you need to help you make the most of these special features.



Programmer's Python: Async

ISBN: 978-1871962595

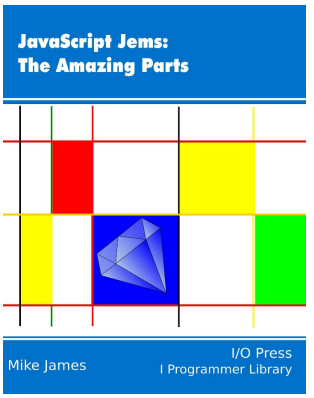
An application that doesn't make use of async code is wasting a huge amount of the machine's potential. Subtitled "Threads, processes, asyncio & more, this volume is about asynchronous programming, something that is is hard to get right, but well worth the trouble and reveals how Python tackles the problems in its own unique way.



Extending & Embedding Python Using C

ISBN: 978-1871962833

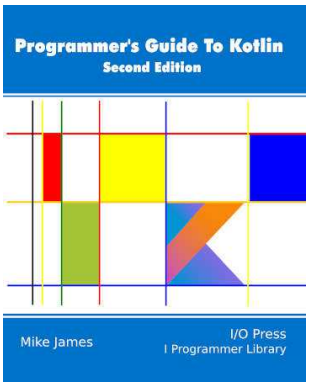
Writing a C extension for Python is good for fun and profit! The fun part is that adding Python to C gives you so much more power and a deeper understanding of how Python works. The internals of Python are worth knowing about because they suggest new approaches to other problems. As well as being interesting, it is also a valuable skill.



JavaScript Jems:The Amazing Parts

ISBN: 978-1871962420

This book is a "meditation" on the features that make JavaScript stand apart from other languages and make it special in terms of having admirable qualities. Each Jem is intended to be an enjoyable read for any JavaScript programmer showing the language in a new light. These are referred to as “Jems”. It’s not a word you will find in the dictionary but it is used in the same sense as its homophone “Gem” as “something prized for its beauty and value”. Here we have a collection of twenty jems about features that have their advantages and disadvantages over their counterparts in other languages.



Programmer’s Guide To Kotlin, Second Edition

ISBN: 978-1871962703

This book introduces Kotlin to programmers. You don't have to be an expert in Java or any other language, but you do need to know the basics of programming and using objects. As with all languages Kotlin has some subtle areas where an understanding of how things work makes all the difference and this second edition pay close attention to these gotchas and has a completely new chapter on Coroutines which is perhaps the Kotlin feature with the most pitfalls and the least documentation. After reading this book all is made clear and workable.