

C + Linux

Are you fed up with the slow speed of your workstation? Would you like to run your programs twice as fast?

Here's how to do it in UNIX. Just follow these three easy steps:

- 1. Design and code hi-performance UNIX vm kernel. Be careful! Your algorithm needs to run twice as fast as the present one to see the full 100% speed.up.*
- 2. Put your code in a file called `/kernel/unix.c`*
- 3. Issue the command*

```
cc -O4 -o /kernel/unix /kernel/unix.c
```

and reboot your machine.

It's as simple as that. And remember, Beethoven wrote his first symphony in C.

—A. P. L. Byteswap's Big Book of Tuning Tips and Rugby Songs

Bitwise operators

A little hardware knowledge is a dangerous thing. One programmer dismantled one of those novelty Christmas cards that plays a carol, and retrieved the piezoelectric melody chip. He secretly installed it into his boss's keyboard, and connected it to one of the LEDs. It turns out that the voltage over a lighted LED is sufficient to drive one of those chips.

Then I, oops, I meant "he", amended the system editor, so it turned on the LED when it started and turned it off when it exited. Result: the boss's terminal played continuous Christmas carol whenever he used the editor! The people in neighboring offices formed a lynch mob after half-an-hour and made him stop all work until the cause was uncovered.

— *The Second Official Handbook of Practical Jokes*

Generally, as a programmer you don't need to concern yourself about operations at the bit level. You're free to think in bytes, or ints and doubles, or even higher level data types composed of a combination of these. But there are times when you'd like to be able to go to the level of an individual bit.

The byte is the lowest level at which we can access data; there's no "bit" type, and we can't ask for an individual bit. In fact, we can't even perform operations on a single bit -- every bitwise operator will be applied to, at a minimum, an entire byte at a time. This means we'll be considering the whole representation of a number whenever we talk about applying a bitwise operator. (Note that this doesn't mean we can't ever change only one bit at a time; it just means we have to be smart about how we do it.) Understanding what it means to apply a bitwise operator to an entire string of bits is probably easiest to see with the shifting operators.

By convention, in C and C++ you can think about binary numbers as starting with the most significant bit to the left (i.e., 10000000 is 128, and 00000001 is 1). Regardless of underlying representation, you may treat this as true. As a consequence, the results of the left and right shift operators are not implementation dependent for unsigned numbers (for signed numbers, the right shift operator is implementation defined).

Shift operators

The leftshift operator is the equivalent of moving all the bits of a number a specified number of places to the left:

```
[variable] << [number of places]
```

For instance, consider the number 8 written in binary 00001000. If we wanted to shift it to the left 2 places, we'd end up with 00100000; everything is moved to the left two places, and zeros are added as padding. This is the number 32 -- in fact, left shifting is the equivalent of multiplying by a power of two.

```
int mult_by_pow_2(int number, int power)
{
    return number<<power;
}
```

Note that in this example, we're using integers, which are either 2 or 4 bytes, and that the operation gets applied to the entire sequence of 16 or 32 bits.

But what happens if we shift a number like 128 and we're only storing it in a single byte: 10000000? Well, $128 * 2 = 256$, and we can't even store a number that big in a byte, so it shouldn't be surprising that the result is 00000000. It shouldn't surprise you that there's a corresponding right-shift operator: \gg (especially considering that I mentioned it earlier). Note that a bitwise right-shift will be the equivalent of integer division by 2.

Why is it integer division? Consider the number 5, in binary, 00000101. $5/2$ is 2.5, but if you are performing integer division, $5/2$ is 2. When you perform a right shift by one: (unsigned int) $5 \gg 1$, you end up with 00000010, as the rightmost 1 gets shifted off the end; this is the representation of the number 2. Note that this only holds true for unsigned integers; otherwise, we are not guaranteed that the padding bits will be all 0s.

Generally, using the left and right shift operators will result in significantly faster code than calculating and then multiplying by a power of two. The shift operators will also be useful later when we look at how to manipulating individual bits.

Bitwise AND

The bitwise AND operator is a single ampersand: $\&$. A handy mnemonic is that the small version of the boolean AND, $\&\&$, works on smaller pieces (bits instead of bytes, chars, integers, etc). In essence, a binary AND simply takes the logical AND of the bits in each position of a number in binary form.

For instance, working with a byte (the char type):

```
01001000 &
10111000 =
-----
00001000
```

The most significant bit of the first number is 0, so we know the most significant bit of the result must be 0; in the second most significant bit, the bit of second number is zero, so we have the same result. The only time where both bits are 1, which is the only time the result will be 1, is the fifth bit from the left. Consequently,

```
72 & 184 = 8
```

Bitwise OR

Bitwise OR works almost exactly the same way as bitwise AND. The only difference is that only one of the two bits needs to be a 1 for that position's bit in the result to be 1. (If both bits are a 1, the result will also have a 1 in that position.) The symbol is a pipe: $|$. Again, this is similar to boolean logical operator, which is $\|\|$.

```
01001000 |
10111000 =
-----
11111000
```

and consequently

Let's take a look at an example of when you could use just these four operators to do something potentially useful. Let's say that you wanted to keep track of certain boolean attributes about something -- for instance, you might have eight cars (!) and want to keep track of which are in use. Let's assign each of the cars a number from 0 to 7.

Since we have eight items, all we really need is a single byte, and we'll use each of its eight bits to indicate whether or not a car is in use. To do this, we'll declare a char called `in_use`, and set it to zero. (We'll assume that none of the cars are initially "in use".)

```
char in_use = 0;
```

Now, how can we check to make sure that a particular car is free before we try to use it? Well, we need to isolate the one bit that corresponds to that car. The strategy is simple: use bitwise operators to ensure every bit of the result is zero except, possibly, for the bit we want to extract.

Consider trying to extract the fifth bit from the right of a number: `XX?XXXXX`. We want to know what the question mark is, and we aren't concerned about the Xs. We'd like to be sure that the X bits don't interfere with our result, so we probably need to use a bitwise AND of some kind to make sure they are all zeros. What about the question mark? If it's a 1, and we take the bitwise AND of `XX?XXXXX` and `00100000`, then the result will be `00100000`:

```
XX1XXXXX &
00100000 =
-----
00100000
```

whereas, if it's a zero, then the result will be `00000000`:

```
XX0XXXXX &
00100000 =
-----
00000000
```

So we get a non-zero number if, and only if, the bit we're interested in is a 1.

This procedure works for finding the bit in the *n*th position. We can use a bitwise leftshift to accomplish this, and it'll be much faster to boot. If we start with the number 1, we are guaranteed to have only a single bit, and we know it's to the far-right. We'll keep in mind that car 0 will have its data stored in the rightmost bit, and car 7 will be the leftmost.

```
int is_in_use(int car_num)
{
    return in_use & 1<<car_num;
}
```

Note that shifting by zero places is a legal operation -- we'll just get back the same number we started with.

All we can do right now is check whether a car is in use; we can't actually set the in-use bit for it. There are two cases to consider: indicating a car is in use, and removing a car from use. In one case, we need to turn a bit on, and in the other, turn a bit off.

Let's tackle the problem of turning the bit on. What does this suggest we should do? If we have a bit set to zero, the only way we know right now to set it to 1 is to do a bitwise OR. Conveniently, if we perform a bitwise OR with only a single bit set to 1 (the rest are 0), then we won't affect the rest of the number because anything ORed with zero remains the same (1 OR 0 is 1, and 0 OR 0 is 0).

Again we need to move a single bit into the correct position:

```
void set_in_use(int car_num)
{
    in_use = in_use | 1<<car_num;
}
```

What does this do? Take the case of setting the rightmost bit to 1: we have some number 0XXXXXXX | 10000000; the result, 1XXXXXXX. The shift is the same as before; the only difference is the operator and that we store the result.

Setting a car to be no longer in use is a bit more complicated. For that, we'll need another operator.

The bitwise complement

The bitwise complement operator, the tilde, `~`, flips every bit. A useful way to remember this is that the tilde is sometimes called a twiddle, and the bitwise complement twiddles every bit: if you have a 1, it's a 0, and if you have a 0, it's a 1.

This turns out to be a great way of finding the largest possible value for an unsigned number:

```
unsigned int max = ~0;
```

0, of course, is all 0s: 00000000 00000000. Once we twiddle 0, we get all 1s: 11111111 11111111. Since max is an unsigned int, we don't have to worry about sign bits or twos complement. We know that all 1s is the largest possible number.

Note that `~` and `!` cannot be used interchangeably. When you take the logical NOT of a non-zero number, you get 0 (FALSE). However, when you twiddle a non-zero number, the only time you'll get 0 is when every bit is turned on. (This non-equivalence principle holds true for bitwise AND too, unless you know that you are using strictly the numbers 1 and 0. For bitwise OR, to be certain that it would be equivalent, you'd need to make sure that the underlying representation of 0 is all zeros to use it interchangeably. But don't do that! It'll make your code harder to understand.)

Now that we have a way of flipping bits, we can start thinking about how to turn off a single bit. We know that we want to leave other bits unaffected, but that if we have a 1 in the given position, we want it to be a 0. Take some time to think about how to do this before reading further.

We need to come up with a sequence of operations that leaves 1s and 0s in the non-target position unaffected; before, we used a bitwise OR, but we can also use a bitwise AND. 1 AND 1 is 1, and 0 AND 1 is 0. Now, to turn off a bit, we just need to AND it with 0: 1 AND 0

is 0. So if we want to indicate that car 2 is no longer in use, we want to take the bitwise AND of XXXXX1XX with 11111011.

How can we get that number? This is where the ability to take the complement of a number comes in handy: we already know how to turn a single bit on. If we turn one bit on and take the complement of the number, we get every bit on except that bit:

```
~(1<<position)
```

Now that we have this, we can just take the bitwise AND of this with the current field of cars, and the only bit we'll change is the one of the car_num we're interested in.

```
int set_unused(int car_num)
{
    in_use = in_use & ~(1<<position);
}
```

You might be thinking to yourself, but this is kind of clunky. We actually need to know whether a car is in use or not (if the bit is on or off) before we can know which function to call. While this isn't necessarily a bad thing, it means that we do need to know a little bit about what's going on. There is an easier way, but first we need the last bitwise operator: exclusive-or.

Bitwise Exclusive-Or (XOR)

There is no boolean operator counterpart to bitwise exclusive-or, but there is a simple explanation. The exclusive-or operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-or, with the operator of a carrot, ^, performs the exclusive-or operation on each pair of bits. Exclusive-or is commonly abbreviated XOR.

For instance, if you have two numbers represented in binary as 10101010 and 01110010 then taking the bitwise XOR results in 11011000. It's easier to see this if the bits are lined up correctly:

```
01110010 ^
10101010
-----
11011000
```

You can think of XOR in the following way: you have some bit, either 1 or 0, that we'll call A. When you take A XOR 0, then you always get A back: if A is 1, you get 1, and if A is 0, you get 0. On the other hand, when you take A XOR 1, you flip A. If A is 0, you get 1; if A is 1, you get 0.

So you can think of the XOR operation as a sort of selective twiddle: if you apply XOR to two numbers, one of which is all 1s, you get the equivalent of a twiddle.

Additionally, if you apply the XOR operation twice -- say you have a bit, A, and another bit B, and you set C equal to A XOR B, and then take C XOR B: you get A XOR B XOR B, which essentially either flips every bit of A twice, or never flips the bit, so you just get back A. (You can also think of B XOR B as canceling out.)

How does that help us? Well, remember the first principle: XORing a bit with 0 results in the same bit. So what we'd really like to be able to do is just call one function that flips the bit of

the car we're interested in -- it doesn't matter if it's being turned on or turned off -- and leaves the rest of the bits unchanged.

This sounds an awful lot like the what we've done in the past; in fact, we only need to make one change to our function to turn a bit on. Instead of using a bitwise OR, we use a bitwise XOR. This leaves everything unchanged, but flips the bit instead of always turning it on:

```
void flip_use_state(int car_num)
{
    in_use = in_use ^ 1<<car_num;
}
```

Further readings

- <http://www.eskimo.com/~scs/cclass/int/sx4ab.html>
- <http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/BitOp/bitwise.html>
- <http://www.edcc.edu/faculty/paul.bladek/Cmpsc142/BitwiseOperations.htm>

Pointers

Never forget that when you point the finger at someone, three of your own fingers are pointing back to you...

—Suspicious spy's proverb

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PCs integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. Further more there is more than one type of integer variable in C. We have integers, long integers and short integers which you can read up on in any basic text on C. This document assumes the use of a 32 bit system with 4 byte integers.

If you want to know the size of the various types of integers on your system, running the following code will give you that information.

```
#include <stdio.h>

int main()
{
    printf("size of a short is %d\n", sizeof(short));
    printf("size of a int is %d\n", sizeof(int));
    printf("size of a long is %d\n", sizeof(long));
}
```

When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name `k` by writing:

```
int k;
```

On seeing the "int" part of this statement the compiler sets aside 4 bytes of memory (on a PC) to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol `k` and the relative address in memory where those 4 bytes were set aside.

Thus, later if we write:

```
k = 2;
```

we expect that, at run time when this statement is executed, the value 2 will be placed in that memory location reserved for the storage of the value of `k`. In C we refer to a variable such as the integer `k` as an "object".

In a sense there are two "values" associated with the object `k`. One is the value of the integer stored there (2 in the above example) and the other the "value" of the memory location, i.e., the address of `k`. Some texts refer to these two values with the nomenclature `rvalue` (right value, pronounced "are value") and `lvalue` (left value, pronounced "el value") respectively.

Now, let's say that we have a reason for wanting a variable designed to hold an lvalue (an address). The size required to hold such a value depends on the system. On older desk top computers with 64K of memory total, the address of any point in memory can be contained in 2 bytes. Computers with more memory would require more bytes to hold an address. The actual size required is not too important so long as we have a way of informing the compiler that what we want to store is an address.

Such a variable is called a pointer variable (for reasons which hopefully will become clearer a little later). In C when we define a pointer variable we do so by preceding its name with an asterisk. In C we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
int *ptr;
```

`ptr` is the name of our variable (just as `k` was the name of our integer variable). The `'*` informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The `int` says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. However, note that when we wrote `int k;` we did not give `k` a value. If this definition is made outside of any function ANSI compliant compilers will initialize it to zero. Similarly, `ptr` has no value, that is we haven't stored an address in it in the above declaration. In this case, again if the declaration is outside of any function, it is initialized to a value guaranteed in such a way that it is guaranteed to not point to any C object or function. A pointer initialized in this manner is called a "null" pointer.

The actual bit pattern used for a null pointer may or may not evaluate to zero since it depends on the specific system on which the code is developed. To make the source code compatible between various compilers on various systems, a macro is used to represent a null pointer. That macro goes under the name `NULL`. Thus, setting the value of a pointer using the `NULL` macro, as with an assignment statement such as `ptr = NULL`, guarantees that the pointer has become a null pointer. Similarly, just as one can test for an integer value of zero, as in `if (k == 0)`, we can test for a null pointer using `if (ptr == NULL)`.

But, back to using our new variable `ptr`. Suppose now that we want to store in `ptr` the address of our integer variable `k`. To do this we use the unary `&` operator and write:

```
ptr = &k;
```

What the `&` operator does is retrieve the lvalue (address) of `k`, even though `k` is on the right hand side of the assignment operator `'='`, and copies that to the contents of our pointer `ptr`. Now, `ptr` is said to "point to" `k`. Bear with us now, there is only one more operator we need to discuss.

The "dereferencing operator" is the asterisk and it is used as follows:

```
*ptr = 7
```

will copy 7 to the address pointed to by `ptr`. Thus if `ptr` "points to" (contains the address of) `k`, the above statement will set the value of `k` to 7. That is, when we use the `'*'` this way we are referring to the value of that which `ptr` is pointing to, not the value of the pointer itself.

Similarly, we could write:

```
printf("%d\n", *ptr);
```

to print to the screen the integer value stored at the address pointed to by `ptr`.

One way to see how all this stuff fits together would be to run the following program and then review the code and the output carefully.

```
#include <stdio.h>

int j, k;
int *ptr;

int main(void)
{
    j = 1;
    k = 2;
    ptr = &k;
    printf("\n");
    printf("j value %d stored at %p\n", j, (void *)&j);
    printf("k value %d stored at %p\n", k, (void *)&k);
    printf("ptr value %p stored at %p\n", ptr, (void *)&ptr);
    printf("The value of the integer pointed to is %d\n", *ptr);

    return 0;
}
```

Pointer types and Arrays

Okay, let's move on. Let us consider why we need to identify the `type` of variable that a pointer points to, as in:

```
int *ptr;
```

One reason for doing this is so that later, once `ptr` "points to" something, if we write:

```
*ptr = 2;
```

the compiler will know how many bytes to copy into that memory location pointed to by `ptr`. If `ptr` was declared as pointing to an integer, 4 bytes would be copied.– Similarly for floats and doubles the appropriate number will be copied. But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code. For example, consider a block in memory consisting of ten integers in a row. That is, 40 bytes of memory are set aside to hold 10 integers.

Now, let's say we point our integer pointer `ptr` at the first of these integers. Furthermore let's say that integer is located at memory location 100 (decimal). What happens when we write:

```
ptr + 1;
```

Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer (its current address, 100, is the address of an integer), it adds 4 to `ptr` instead of 1, so the pointer "points to" the `next integer`, at memory location 104. Similarly, were the `ptr` declared as a pointer to a short, it would add 2 to it instead of 1. The same goes for other data types such as floats, doubles, or even user defined data types such as structures. This is obviously not the same kind of "addition" that we normally think of. In C it is referred to as addition using "pointer arithmetic", a term which we will come back to later.

Similarly, since `++ptr` and `ptr++` are both equivalent to `ptr + 1` (though the point in the program when `ptr` is incremented may be different), incrementing a pointer using the unary `++` operator, either pre- or post-, increments the address it stores by the amount `sizeof(type)` where "type" is the type of the object pointed to. (i.e. 4 for an integer).

Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers.

Consider the following:

```
int my_array[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to `my_array`, i.e. using `my_array[0]` through `my_array[5]`. But, we could alternatively access them via a pointer as follows:

```
int *ptr;
ptr = &my_array[0];
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;
    ptr = &my_array[0];

    printf("\n\n");
    for (i = 0; i < 6; i++)
    {
        printf("my_array[%d] = %d    ",i,my_array[i]);
        printf("ptr + %d = %d\n",i, *(ptr + i));
    }
    return 0;
}
```

In C, the standard states that wherever we might use `&var_name[0]` we can replace that with `var_name`, thus in our code where we wrote:

```
ptr = &my_array[0];
```

we can write:

```
ptr = my_array;
```

to achieve the same result. This leads many texts to state that the name of an array is a pointer but it's really not.

Also we have learned that on different systems the size of a pointer can vary. As it turns out it is also possible that the size of a pointer can vary depending on the data type of the object to which it points. Thus, as with integers where you can run into trouble attempting to assign a long integer to a variable of type short integer, you can run into trouble attempting to assign the values of pointers of various types to pointer variables of other types.

To minimize this problem, C provides for a pointer of type void. We can declare such a pointer by writing:

```
void *vptr;
```

A void pointer is sort of a generic pointer. For example, while C will not permit the comparison of a pointer to type integer with a pointer to type character, for example, either of these can be compared to a void pointer. Of course, as with other variables, casts can be used to convert from one type of pointer to another under the proper circumstances.

Pointers and Structures

As you may know, we can declare the form of a block of data containing different data types by means of a structure declaration. For example, a personnel file might contain structures which look something like:

```
struct tag {
    char lname[20];        /* last name */
    char fname[20];       /* first name */
    int age;               /* age */
    float rate;           /* e.g. 12.75 per hour */
};
```

Let's say we have a bunch of these structures in a disk file and we want to read each one out and print out the first and last name of each one so that we can have a list of the people in our files. The remaining information will not be printed out. We will want to do this printing with a function call and pass to that function a pointer to the structure at hand. For demonstration purposes I will use only one structure for now. But realize the goal is the writing of the function, not the reading of the file which, presumably, we know how to do.

For review, recall that we can access structure members with the dot operator as in:

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

struct tag {
    char lname[20];
```

```

    char fname[20];
    int age;
    float rate;
};

struct tag my_struct;

int main(void)
{
    strcpy(my_struct.lname, "Jensen");
    strcpy(my_struct.fname, "Ted");
    printf("\n%s ", my_struct.fname);
    printf("%s\n", my_struct.lname);
    return 0;
}

```

Now, this particular structure is rather small compared to many used in C programs. To the above we might want to add:

```

date_of_hire;                (data types not shown)
date_of_last_raise;
last_percent_increase;
emergency_phone;
medical_plan;
Social_S_Nbr;
etc.....

```

If we pass the whole structure it means that we must copy the contents of the structure from the calling function to the called function. In systems using stacks, this is done by pushing the contents of the structure on the stack. With large structures this could prove to be a problem. However, passing a pointer uses a minimum amount of stack space.

In any case, since this is a discussion of pointers, we will discuss how we go about passing a pointer to a structure and then using it within the function.

Consider the case described, i.e. we want a function that will accept as a parameter a pointer to a structure and from within that function we want to access members of the structure. For example we want to print out the name of the employee in our example structure.

Okay, so we know that our pointer is going to point to a structure declared using struct tag. We declare such a pointer with the declaration:

```
struct tag *st_ptr;
```

and we point it to our example structure with:

```
st_ptr = &my_struct;
```

Now, we can access a given member by de-referencing the pointer. But, how do we de-reference the pointer to a structure? Well, consider the fact that we might want to use the pointer to set the age of the employee. We would write:

```
(*st_ptr).age = 63;
```

Look at this carefully. It says, replace that within the parenthesis with that which `st_ptr` points to, which is the structure `my_struct`. Thus, this breaks down to the same as `my_struct.age`.

However, this is a fairly often used expression and the designers of C have created an alternate syntax with the same meaning which is:

```
st_ptr->age = 63;
```

With that in mind, look at the following program:

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

struct tag{
    char lname[20];
    char fname[20];
    int age;
    float rate;
};

struct tag my_struct;
void show_name(struct tag *p);

int main(void)
{
    struct tag *st_ptr;
    st_ptr = &my_struct;
    strcpy(my_struct.lname, "Jensen");
    strcpy(my_struct.fname, "Ted");
    printf("\n%s ", my_struct.fname);
    printf("%s\n", my_struct.lname);
    my_struct.age = 63;
    show_name(st_ptr);
    return 0;
}

void show_name(struct tag *p)
{
    printf("\n%s ", p->fname);
    printf("%s ", p->lname);
    printf("%d\n", p->age);
}
```

Pointers to functions

A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say

```
int func(int a, float b);
```

and simply put brackets around the name and a * in front of it: that declares the pointer. Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* function returning pointer to int */  
int *func(int a, float b);
```

```
/* pointer to function returning int */  
int (*func)(int a, float b);
```

Once you've got the pointer, you can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression. You can call the function using one of two forms:

```
(*func)(1,2);  
/* or */  
func(1,2);
```

The second form has been newly blessed by the Standard. Here's a simple example.

```
#include <stdio.h>  
#include <stdlib.h>  
  
void func(int);  
  
main(){  
    void (*fp)(int);  
  
    fp = func;  
  
    (*fp)(1);  
    fp(2);  
  
    exit(EXIT_SUCCESS);  
}  
  
void  
func(int arg){  
    printf("%d\n", arg);  
}
```

If you like writing finite state machines, you might like to know that you can have an array of pointers to functions, with declaration and use like this:

```
void (*fparr[])(int, float) = {  
    /* initializers */  
};
```

```
/* then call one */
```

```
fparr[5](1, 3.4);
```

Further readings

- <http://pw1.netcom.com/~tjensen/ptr/pointers.htm>
- <http://www.augustcouncil.com/~tgibson/tutorial/ptr.html>
- <http://oreilly.com/catalog/pcp3/chapter/ch13.html>

Organizing code files

Should arrays indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration.

—Stan Kelly-Bootle

There are benefits to splitting up your project into several smaller files. So, how would you go about it? Although some of the decisions you make will be reasonably arbitrary, there are some basic rules that you should follow to ensure that it all works.

Firstly, look at how you would split your code into sections. Often this is by splitting it into separate subsystems, or 'modules', such as sound, music, graphics, file handling, etc. Create new files with meaningful filenames so that you know at a glance what kind of code is in them. Then move all the code that belongs to that module into that file. Sometimes you don't have clear module - some would say this should send out warnings about the quality of your design! There may still be other criteria you use for splitting up code, such as the structures it operates upon. (Often "general purpose" functions can be split into string-handling and number-handling, for example.) And occasionally a module could be split into two or more files, because it might make sense to do so on logical grounds.

Once you have split it up in this way into separate source files, the next stage is to consider what will go into the header files. On a very simple level, code that you usually put at the top of the source file is a prime candidate for moving into a separate header file. This is presumably why they got termed 'header' files, after all.

This code to go in a header usually includes some or all of the following:

- ▶ class and struct definitions
- ▶ typedefs
- ▶ function prototypes
- ▶ global variables (but see below)
- ▶ constants
- ▶ #defined macros
- ▶ #pragma directives

You generally want one header file for every source file. That is, a SPRITES.CPP probably needs a SPRITES.H file, a SOUND.CPP needs a SOUND.H, and so on. Keep the naming consistent so that you can instantly tell which header goes with which normal file.

These header files become the interface between your subsystems. By #including a header, you gain access to all the structure definitions, function prototypes, constants etc for that subsystem. Therefore, every source file that uses sprites in some way will probably have to #include "sprite.h", every source file that uses sound may need to #include "sound.h", and so on. Note that you use quotes rather than angular brackets when #including your own files. The quotes tell the compiler to look for your headers in the program directory first, rather than the compiler's standard headers.

Remember that, as far as the compiler is concerned, there is absolutely no difference between a header file and a source file. (Exception: some compilers will refuse to compile a header file directly, assuming you made a mistake in asking.) As stated earlier, they are both just plain text files that are filled with code. The distinction is a conceptual one that programmers must adhere to in order to keep the logical file structure intact. The key idea is

that headers contain the interface, and the source files contain the actual implementation. This applies whether you are working in C or C++, in an object-oriented way or a structural way. This means that one source file uses another source file via the second source file's header.

Problems

The rules given above are fairly vague and merely serve as a starting point for organizing your code. In simple cases, you can produce completely working programs by following those guidelines. However there are some more details that have to be accounted for, and it is often these details that cause novice programmers so much grief when they first start splitting their code up into header files and normal files.

In my experience, there are four basic errors that people encounter when they first enter the murky world of user-defined header files.

1. The source files no longer compile as they can't find the functions or variables that they need.
2. Cyclic dependencies, where headers appear to need to `#include` each other to work as intended. A `Sprite` may contain a pointer to the `Creature` it represents, and a `Creature` may contain a pointer to the `Sprite` it uses. No matter how you do this, either `Creature` or `Sprite` must be declared first in the code, and that implies that it won't work since the other type isn't declared yet.
3. Duplicate definitions where a class or identifier is included twice in a source file. This is a compile time error and usually arises when multiple header files include one other header file, leading to that header being included twice when you compile a source file that uses them.
4. Duplicate instances of objects within the code that compiled fine. This is a linking error, often difficult to understand.

Fixing problem 1

The first error, where a source file refuses to compile because one of the identifiers was undeclared, is easy to resolve. Simply `#include` the file that contains the definition of the identifier you need. If your header files are organized logically and named well, this should be easy. If you need to use the `Sprite` struct, then you probably need to `#include "sprite.h"` in every file that does so. One mistake that programmers often make is to assume that a file is `#included` simply because another header `#includes` it for itself.

```
/* Header1.h */
#include "header2.h"
class ClassOne { ... };

/* Header2.h */
class ClassTwo { ... };

/* File1.cpp */
#include "Header1.h"
```

```
ClassOne myClassOne_instance;
ClassTwo myClassTwo_instance;
```

In this case, File1.cpp will compile fine, as including Header1.h has indirectly #included Header2.h, meaning that File1.cpp has access to the Class2 class. But what happens if, at a later date, someone realises that Header1.h doesn't actually need to #include Header2.h? They can remove that #include line, and suddenly File1.cpp will break the next time you try to compile it.

The key here, is to explicitly #include any header files that you need for a given source file to compile. You should never rely on header files indirectly including extra headers for you, as that may change. The 'extra' #includes also serve as documentation, by demonstrating what other code this file is dependent on. So don't try and leave them out if you know you need that header included somehow.

Fixing problem 2

Cyclic (or two-way) dependencies are a common problem in software engineering. Many constructs involve a two-way link of some sort, and this implies that both classes or structures know about each other. Often this ends up looking like this:

```
#include "child.h"
class Parent
{
    Child* theChild;
};

/* Child.h */
#include "parent.h"
class Child
{
    Parent* theParent;
};
```

Given that one of these has to be compiled first, you need some way to break the cycle. In this case, it's actually quite trivial. The Parent struct doesn't actually need to know the details of the Child class, as it only stores a pointer to one. Pointers are pretty much the same no matter what they point to, therefore you don't need to the definition of the structure or class in order to store a pointer to an instance of that structure or class. So the #include line is not needed. However, simply taking it out will give you an "undeclared identifier" error when it encounters the word 'Child', so you need to let the compiler know that Child is a class or class that you wish to point to. This is done with a forward declaration, taking the form of a class or class definition without a body. Example:

```
/* Parent.h */
class Child; /* Forward declaration of Child; */
class Parent
{
    Child* theChild;
};
```

Notice how the `#include` line is replaced by the forward declaration. This has allowed you to break the dependency between `Parent.h` and `Child.h`. Additionally, it will speed up compilation as you are reading in one less header file. In this case, the same procedure can (and should) be followed in `Child.h` by forward declaring `"class Parent;"`. As long as you are only referring to a pointer and not the actual type itself, you don't need to `#include` the full definition. In 99% of cases, this can be applied to one or both sides of a cycle to remove the need to `#include` one header from another, eliminating the cyclic dependency.

Of course, in the source files, it's quite likely that there will be functions that apply to `Parent` that will manipulate the `Child` also, or vice versa. Therefore, it is probably necessary to `#include` both `parent.h` and `child.h` in `parent.c` and `child.c`.

Fixing problem 3

Duplicate definitions at compile time imply that a header ended up being included more than once for a given source file. This leads to a class or struct being defined twice, causing an error. The first thing you should do is ensure that, for each source file, you only include the headers that you need. This will aid compilation speed since the compiler is not reading and compiling headers that serve no purpose for this file.

Sadly, this is rarely enough, since some headers will include other headers. Let's revisit an example from earlier, slightly modified:

```
/* Header1.h */
#include "header3.h"
class ClassOne { ... };

/* Header2.h */
#include "header3.h"
class ClassTwo { ... };

/* File1.cpp */
#include "Header1.h"
#include "Header2.h"
ClassOne myClassOne_instance;
ClassTwo myClassTwo_instance;
```

`Header1.h` and `Header2.h` `#include` `header3.h` for some reason. Maybe `ClassOne` and `ClassTwo` are composed out of some class defined in `Header3.h`. The reason itself is not important; the point is that sometimes headers include other headers without you explicitly asking for it, which means that sometimes a header will be `#included` twice for a given source file, despite your best intentions. Note that it doesn't matter that `header3.h` is being `#included` from different header files: during compilation, these all resolve to one file. The `#include` directive literally says "include the specified file right here in this file while we process it", so all the headers get dumped inline into your source file before it gets compiled.

For the purposes of compilation, `File1.cpp` ends up containing copies of `Header1.h` and `Header2.h`, both of which include their own copies of `Header3.h`. The resulting file, with all headers expanded inline into your original file, is known as a translation unit. Due to this inline expansion, anything declared in `Header3.h` is going to appear twice in this translation unit, causing an error.

So, what do you do? You can't do without Header1.h or Header2.h, since you need to access the structures declared within them. So you need some way of ensuring that, no matter what, Header3.h is not going to appear twice in your File1.cpp translation unit when it gets compiled. This is where inclusion guards come in.

If you looked at `stdlib.h` earlier, you may have noticed lines near the top similar to the following:

```
#ifndef _INC_STDLIB
#define _INC_STDLIB
```

And at the bottom of the file, something like:

```
#endif /* _INC_STDLIB */
```

This ensures that the 'do something' only ever happens once. During compilation of File1.cpp, the first time any file asks to `#include` `stdlib.h`, it reaches the `#ifndef` line and continues because `"_INC_STDLIB"` is not yet defined. The very next line defines that symbol and carries on reading in `stdlib.h`. If there is another `"#include "` during the compilation of File1.cpp, it will read to the `#ifndef` check and then skip to the `#endif` at the end of the file. This is because everything between the `#ifndef` and the `#endif` is only executed if `"_INC_STDLIB"` is not defined; and it got defined the first time we included it. This way, it is ensured that the definitions within `stdlib.h` are only ever included once by putting them within this `#ifndef / #endif` pair.

This is trivial to apply to your own projects. At the start of every header file you write, put the following:

```
#ifndef INC_FILENAME_H
#define INC_FILENAME_H
```

Note that the symbol (in this case, `"INC_FILENAME_H"`) needs to be unique across your project. This is why it is a good idea to incorporate the filename into the symbol. Don't add an underscore at the start like `stdlib.h` does, as identifiers prefixed with an underscore are supposed to be reserved for "the implementation" (ie. the compiler, the standard libraries, and so on). Then add the `#endif /* INC_FILENAME_H */` at the end of the file. The comment is not necessary, but will help you remember what that `#endif` is there for.

Fixing problem 4

See next chapter.

Further readings

- http://www.andrewnoske.com/wiki/index.php?title=Code_-_organizing_files_in_c
- http://gcc.gnu.org/onlinedocs/gcc-3.0.1/cpp_2.htm
- <http://www.physics.irfu.se/aCCdoc/precomp.htm>

Storage class specifiers

The three most dangerous things in the world are a programmer with a soldering iron, a hardware type with a program patch and a user with an idea."

— *_The Wizardry Compiled_ by Rick Cook*

There are five keywords under the category of storage class specifiers, although one of them, `typedef`, is there more out of convenience than utility; it has its own section later since it doesn't really belong here. The ones remaining are `auto`, `extern`, `register`, and `static`.

Storage class specifiers help you to specify the type of storage used for data objects. Only one storage class specifier is permitted in a declaration—this makes sense, as there is only one way of storing things—and if you omit the storage class specifier in a declaration, a default is chosen. The default depends on whether the declaration is made outside a function (external declarations) or inside a function (internal declarations). For external declarations the default storage class specifier will be `extern` and for `internal` declarations it will be `auto`. The only exception to this rule is the declaration of functions, whose default storage class specifier is always `extern`.

For a full understanding, you need a good grasp of three distinct but related concepts. The Standard calls them:

- ▶ duration
- ▶ scope
- ▶ linkage

Duration

The duration of an object describes whether its storage is allocated once only, at program start-up, or is more transient in its nature, being allocated and freed as necessary.

There are only two types of duration of objects: static duration and automatic duration. Static duration means that the object has its storage allocated permanently, automatic means that the storage is allocated and freed as necessary. Data objects declared inside functions are given the default storage class specifier of `auto` unless some other storage class specifier is used.

```
#include <stdio.h>
#define MAX 5

void sumIt(void);

int main()
{
    int i = 0;
    printf("Enter 5 numbers to be summed\n");

    for(i = 0; i < MAX; ++i)
```

```

        sumIt();

    printf("Program completed\n");

    return 0;
}

void sumIt(void)
{
    static int sum = 0;
    int num;

    printf("\nEnter a number: ");
    scanf("%d", &num);

    sum+=num;

    printf("The current sum is: %d\n",sum);
}

```

The `register` storage class is quite interesting, although it is tending to fall into disuse nowadays. It suggests to the compiler that it would be a good idea to store the object in one or more hardware registers in the interests of speed. The compiler does not have to take any notice of this, but to make things easy for it, register variables do not have an address (the `&` address-of operator is forbidden) because some computers don't support the idea of addressable registers.

Scope

Now we must look again at the scope of the names of objects, which defines when and where a given name has a particular meaning. The different types of scope are the following:

- ▶ function scope
- ▶ file scope
- ▶ block scope
- ▶ function prototype scope

The easiest is function scope. This only applies to labels, whose names are visible throughout the function where they are declared, irrespective of the block structure.

Any name declared outside a function has file scope, which means that the name is usable at any point from the declaration on to the end of the source code file containing the declaration. Of course it is possible for these names to be temporarily hidden by declarations within compound statements.

A name declared inside a compound statement, or as a formal parameter to a function, has block scope and is usable up to the end of the associated `}` which closes the compound statement. Any declaration of a name within a compound statement hides any outer declaration of the same name until the end of the compound statement.

A special and rather trivial example of scope is function prototype scope where a declaration of a name extends only to the end of the function prototype.

```

#include <stdio.h>

int i = 1;

int main(int argc, char * argv[])
{
    printf("%d\n", i);
    {
        int i = 2, j = 3;
        printf("%d\n%d\n", i, j);
        {
            int i = 0;
            printf("%d\n%d\n", i, j);
        }
        printf("%d\n", i);
    }
    printf("%d\n", i);
    return 0;
}

```

Linkage

Linkage is used to determine what makes the same name declared in different scopes refer to the same thing. An object only ever has one name, but in many cases we would like to be able to refer to the same object from different scopes.

The three different types of linkage are:

- ▶ external linkage
- ▶ internal linkage
- ▶ no linkage

In an entire program, built up perhaps from a number of source files and libraries, if a name has external linkage, then every instance of that name refers to the same object throughout the program. For something which has internal linkage, it is only within a given source code file that instances of the same name will refer to the same thing. Finally, names with no linkage refer to separate things.

The rules that determine the linkage and definition associated with declarations look quite complicated. The three types of accessibility that you will want of data objects or functions are:

- ▶ throughout the entire program
- ▶ restricted to one source file
- ▶ restricted to one function

The recommended practice for the first two cases is to declare all of the names in each of the relevant source files before you define any functions.

- ▶ external linkage declarations

- ▶ internal linkage declarations
- ▶ functions

The external linkage declarations would be prefixed with `extern`, the internal linkage declarations with `static`.

```
/* example of a single source file layout */
#include <stdio.h>

/* Things with external linkage:
 * accessible throughout program.
 * These are declarations, not definitions, so
 * we assume their definition is somewhere else.
 */

extern int important_variable;
extern int library_func(double, int);

/*
 * Definitions with external linkage.
 */
extern int ext_int_def = 0;      /* explicit definition */
int tent_ext_int_def;          /* tentative definition */

/*
 * Things with internal linkage:
 * only accessible inside this file.
 * The use of static means that they are also
 * tentative definitions.
 */

static int less_important_variable;
static struct{
    int member_1;
    int member_2;
}local_struct;

/*
 * Also with internal linkage, but not a tentative
 * definition because this is a function.
 */
static void lf(void);

/*
 * Definition with internal linkage.
 */
static float int_link_f_def = 5.3;

/*
 * Finally definitions of functions within this file
 */
```

```

/*
 * This function has external linkage and can be called
 * from anywhere in the program.
 */
void f1(int a){

/*
 * These two functions can only be invoked by name from
 * within this file.
 */
static int local_function(int a1, int a2){
    return(a1 * a2);
}

static void lf(void){
    /*
     * A static variable with no linkage,
     * so usable only within this function.
     * Also a definition (because of no linkage)
     */
    static int count;
    /*
     * Automatic variable with no linkage but
     * an initializer
     */
    int i = 1;

    printf("lf called for time no %d\n", ++count);
}
/*
 * Actual definitions are implicitly provided for
 * all remaining tentative definitions at the end of
 * the file
 */

```

If the program is in several source files, and a variable is defined in file1 and used in file2 and file3, then extern declarations are needed in file2 and file3 to connect the occurrences of the variable. The usual practice is to collect extern declarations of variables and functions in a separate file, historically called a header.

In file1:

```

int GlobalVariable;
void SomeFunction();
int main(){
    GlobalVariable = 1;
    SomeFunction();
    return(0);
}

```

In file2:

```
extern int GlobalVariable;
void SomeFunction(){
    ++GlobalVariable;
}
```

Further readings

- http://icecube.wisc.edu/~dglo/c_class/vstorage.html
- <http://www.eetimes.com/discussion/programming-pointers/4026823/Storage-class-specifiers-and-storage-duration>

Type qualifiers

The primary purpose of the DATA statement is to give names to constants; instead of referring to pi as 3.141592653589793 at every appearance, the variable PI can be given that value with a DATA statement and used instead of the longer form of the constant. This also simplifies modifying the program, should the value of pi change."

— FORTRAN manual for Xerox computers

The keywords `const` and `volatile` can be applied to any declaration, including those of structures, unions, enumerated types or typedef names. Applying them to a declaration is called qualifying the declaration.

Const

Let's look at what is meant when `const` is used. It's really quite simple: `const` means that something is not modifiable, so a data object that is declared with `const` as a part of its type specification must not be assigned to in any way during the run of a program. For example, if you were accessing a hardware port at a fixed memory address and promised only to read from it, then it would be declared to be `const` but not initialized.

Taking the address of a data object of a type which isn't `const` and putting it into a pointer to the `const`-qualified version of the same type is both safe and explicitly permitted; you will be able to use the pointer to inspect the object, but not modify it. Putting the address of a `const` type into a pointer to the unqualified type is much more dangerous and consequently prohibited (although you can get around this by using a cast). Here is an example:

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

main(){
    int i;
    const int ci = 123;

    /* declare a pointer to a const.. */
    const int *cpi;

    /* ordinary pointer to a non-const */
    int *ncpi;

    cpi = &ci;
    ncpi = &i;

    /*
     * this is allowed
     */
    cpi = ncpi;

    /*
     * this needs a cast
     * because it is usually a big mistake,
     * see what it permits below.
    */
}
```

```

    */
    ncpi = (int *)cpi;

    /*
     * now to get undefined behaviour...
     * modify a const through a pointer
     */
    *ncpi = 0;

    exit(EXIT_SUCCESS);
}

```

As the example shows, it is possible to take the address of a constant object, generate a pointer to a non-constant, then use the new pointer. This is an error in your program and results in undefined behaviour.

The main intention of introducing const objects was to allow them to be put into read-only store, and to permit compilers to do extra consistency checking in a program. Unless you defeat the intent by doing naughty things with pointers, a compiler is able to check that const objects are not modified explicitly by the user.

An interesting extra feature pops up now. What does this mean?

```

char c;
char *const cp = &c;

```

It's simple really; `cp` is a pointer to a char, which is exactly what it would be if the const weren't there. The const means that `cp` is not to be modified, although whatever it points to can be—the pointer is constant, not the thing that it points to. The other way round is

```

const char *cp;

```

which means that now `cp` is an ordinary, modifiable pointer, but the thing that it points to must not be modified. So, depending on what you choose to do, both the pointer and the thing it points to may be modifiable or not; just choose the appropriate declaration.

Volatile

The reason for having this type qualifier is mainly to do with the problems that are encountered in real-time or embedded systems programming using C. Imagine that you are writing code that controls a hardware device by placing appropriate values in hardware registers at known absolute addresses.

Let's imagine that the device has two registers, each 16 bits long, at ascending memory addresses; the first one is the control and status register (csr) and the second is a data port. The traditional way of accessing such a device is like this:

```

/* Standard C example but without const or volatile */
/*
 * Declare the device registers
 * Whether to use int or short
 * is implementation dependent
 */

```

```

struct devregs{
    unsigned short  csr;    /* control & status */
    unsigned short  data;  /* data port */
};

/* bit patterns in the csr */
#define ERROR      0x1
#define READY     0x2
#define RESET     0x4

/* absolute address of the device */
#define DEVADDR ((struct devregs *)0xffff0004)

/* number of such devices in system */
#define NDEVS     4

/*
 * Busy-wait function to read a byte from device n.
 * check range of device number.
 * Wait until READY or ERROR
 * if no error, read byte, return it
 * otherwise reset error, return 0xffff
 */
unsigned int read_dev(unsigned devno){

    struct devregs *dvp = DEVADDR + devno;

    if(devno >= NDEVS)
        return(0xffff);

    while((dvp->csr & (READY | ERROR)) == 0)
        ; /* NULL - wait till done */

    if(dvp->csr & ERROR){
        dvp->csr = RESET;
        return(0xffff);
    }

    return((dvp->data) & 0xff);
}

```

However, a major problem with previous C compilers would be in the while loop which tests the status register and waits for the **ERROR** or **READY** bit to come on. Any self-respecting optimizing compiler would notice that the loop tests the same memory address over and over again. It would almost certainly arrange to reference memory once only, and copy the value into a hardware register, thus speeding up the loop. This is, of course, exactly what we don't want; this is one of the few places where we must look at the place where the pointer points, every time around the loop.

Because of this problem, most C compilers have been unable to make that sort of optimization in the past. To remove the problem (and other similar ones to do with when to

write to where a pointer points), the keyword volatile was introduced. It tells the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the program itself, and forces every reference to such an object to be a genuine reference. Here is how you would rewrite the example, making use of const and volatile to get what you want.

```
/*
 * Declare the device registers
 * Whether to use int or short
 * is implementation dependent
 */

struct devregs{
    unsigned short volatile csr;
    unsigned short const volatile data;
};

/* bit patterns in the csr */
#define ERROR    0x1
#define READY    0x2
#define RESET    0x4

/* absolute address of the device */
#define DEVADDR ((struct devregs *)0xffff0004)

/* number of such devices in system */
#define NDEVS    4

/*
 * Busy-wait function to read a byte from device n.
 * check range of device number.
 * Wait until READY or ERROR
 * if no error, read byte, return it
 * otherwise reset error, return 0xffff
 */
unsigned int read_dev(unsigned devno){
    struct devregs * const dvp = DEVADDR + devno;

    if(devno >= NDEVS)
        return(0xffff);

    while((dvp->csr & (READY | ERROR)) == 0)
        ; /* NULL - wait till done */

    if(dvp->csr & ERROR){
        dvp->csr = RESET;
        return(0xffff);
    }

    return((dvp->data) & 0xff);
}
```

Further readings

- <http://www.eskimo.com/~scs/cclass/int/sx4ga.html>
- <http://docs.sun.com/app/docs/doc/820-7598/bjakl?a=view>
- <http://knol.google.com/k/volatile-variable-c-language#>

Libraries

One of the main causes of the fall of the Roman Empire was that—lacking zero—they had no way to indicate successful termination of their C programs.

— Robert Firth

In computer science, a library is a collection of subroutines or classes used to develop software.

Libraries contain code and data that provide services to independent programs. This allows the sharing and changing of code and data in a modular fashion. Some executables are both standalone programs and libraries, but most libraries are not executables. Executables and libraries make references known as links to each other through the process known as linking, which is typically done by a linker.

Static libraries

Static libraries are simply a collection of ordinary object files; conventionally, static libraries end with the `.a` suffix. This collection is created using the `ar` (archiver) program. Static libraries aren't used as often as they once were, because of the advantages of shared libraries (described below). Still, they're sometimes created, they existed first historically, and they're simpler to explain.

Static libraries permit users to link to programs without having to recompile its code, saving recompilation time. Note that recompilation time is less important given today's faster compilers, so this reason is not as strong as it once was. Static libraries are often useful for developers if they wish to permit programmers to link to their library, but don't want to give the library source code (which is an advantage to the library vendor, but obviously not an advantage to the programmer trying to use the library). In theory, code in static ELF libraries that is linked into an executable should run slightly faster (by 1-5%) than a shared library or a dynamically loaded library, but in practice this rarely seems to be the case due to other confounding factors.

The standard system libraries are usually found in the directories `/usr/lib` and `/lib`. For example, the C math library is typically stored in the file `/usr/lib/libm.a` on Unix-like systems. The corresponding prototype declarations for the functions in this library are given in the header file `/usr/include/math.h`. The C standard library itself is stored in `/usr/lib/libc.a` and contains functions specified in the ANSI/ISO C standard, such as `printf`---this library is linked by default for every C program.

Here is an example program which makes a call to the external function `sqrt` in the math library `libm.a`:

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

int
main (void)
{
    double x = sqrt (2.0);
```

```
    printf ("The square root of 2.0 is %f\n", x);
    return 0;
}
```

Trying to create an executable from this source file alone causes the compiler to give an error at the link stage:

```
$ gcc -Wall calc.c -o calc
/tmp/ccbR6Ojm.o: In function `main':
/tmp/ccbR6Ojm.o(.text+0x19): undefined reference to `sqrt'
```

The problem is that the reference to the `sqrt` function cannot be resolved without the external math library `'libm.a'`. The function `sqrt` is not defined in the program or the default library `'libc.a'`, and the compiler does not link to the file `'libm.a'` unless it is explicitly selected. Incidentally, the file mentioned in the error message `'/tmp/ccbR6Ojm.o'` is a temporary object file created by the compiler from `'calc.c'`, in order to carry out the linking process.

To enable the compiler to link the `sqrt` function to the main program `'calc.c'` we need to supply the library `'libm.a'`. One obvious but cumbersome way to do this is to specify it explicitly on the command line:

```
$ gcc -Wall calc.c /usr/lib/libm.a -o calc
```

The library `'libm.a'` contains object files for all the mathematical functions, such as `sin`, `cos`, `exp`, `log` and `sqrt`. The linker searches through these to find the object file containing the `sqrt` function.

Once the object file for the `sqrt` function has been found, the main program can be linked and a complete executable produced:

```
$ ./calc
The square root of 2.0 is 1.414214
```

The executable file includes the machine code for the main function and the machine code for the `sqrt` function, copied from the corresponding object file in the library `'libm.a'`.

To avoid the need to specify long paths on the command line, the compiler provides a short-cut option `'-l'` for linking against libraries. For example, the following command:

```
$ gcc -Wall calc.c -lm -o calc
```

is equivalent to the original command above using the full library name `'/usr/lib/libm.a'`.

In general, the compiler option `-lNAME` will attempt to link object files with a library file `'libNAME.a'` in the standard library directories. Additional directories can be specified with command-line options and environment variables, to be discussed shortly. A large program will typically use many `-l` options to link libraries such as the math library, graphics libraries and networking libraries.

Dynamic/shared libraries

Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. It's actually much more flexible and sophisticated than this, because the approach used by Linux permits you to:

- ▶ update libraries and still support programs that want to use older, non-backward-compatible versions of those libraries;
- ▶ override specific libraries or even specific functions in a library when executing a particular program.
- ▶ do all this while programs are running using existing libraries.

Every shared library has a special name called the “soname”. The soname has the prefix “lib”, the name of the library, the phrase “.so”, followed by a period and a version number that is incremented whenever the interface changes (as a special exception, the lowest-level C libraries don't start with “lib”).

Every shared library also has a “real name”, which is the filename containing the actual library code. The real name adds to the soname a period, a minor number, another period, and the release number. The last period and release number are optional. The minor number and release number support configuration control by letting you know exactly what version(s) of the library are installed. Note that these numbers might not be the same as the numbers used to describe the library in documentation, although that does make things easier.

In addition, there's the name that the compiler uses when requesting a library, (I'll call it the “linker name”), which is simply the soname without any version number.

What this boils down to is this: if the linker finds that the definition for a particular symbol is in a shared library, then it doesn't include the definition of that symbol in the final executable. Instead, the linker records the name of symbol and which library it is supposed to come from in the executable file instead.

When the program is run, the operating system arranges that these remaining bits of linking are done “just in time” for the program to run. Before the main function is run, a smaller version of the linker (often called `ld.so`) goes through these promissory notes and does the last stage of the link there and then—pulling in the code of the library and joining up all of the dots.

There's another big difference with how shared libraries work compared to static libraries, and that shows up in the granularity of the link. If a particular symbol is pulled in from a particular shared library (say `printf` in `libc.so`), then the whole of that shared library is mapped into the address space of the program. This is very different from the behavior of a static library, where only the particular objects that held undefined symbols got pulled in.

Compiling for runtime linking with a dynamically linked `libctest.so.1.0`:

```
gcc -Wall -L/opt/lib prog.c -lctest -o prog
```

Where the name of the library is `libctest.so`.

The libraries will NOT be included in the executable but will be dynamically linked during runtime execution.

In order for an executable to find the required libraries to link with during run time, one must configure the system so that the libraries can be found. Methods available:

1. Add library directories to be included during dynamic linking to the file `/etc/ld.so.conf`. Add the library path to this file and then execute the command (as root) `ldconfig` to configure the linker run-time bindings.

2. Add specified directory to library cache: (as root) `ldconfig -n /opt/lib` where `/opt/lib` is the directory containing your library `libctest.so`
(When developing and just adding your current directory: `ldconfig -n. Link -L.`)

3. Specify the environment variable `LD_LIBRARY_PATH` to point to the directory paths containing the shared object library. This will specify to the run time loader that the library paths will be used during execution to resolve dependencies.

Further readings

- <http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html>
- <http://docs.sun.com/source/819-3690/Building.Libs.html>
- <http://www.faqs.org/docs/Linux-HOWTO/Program-Library-HOWTO.html>

Stack

It's 5.50 a.m.... Do you know where your stack pointer is ?

— Anonymous

Variables represent storage space in the computer's memory. Each variable presents a convenient names like `length` or `sum` in the source code. Behind the scenes at runtime, each variable uses an area of the computer's memory to store its value. It is not the case that every variable in a program has a permanently assigned area of memory. Instead, modern languages are smart about giving memory to a variable only when necessary. The terminology is that a variable is allocated when it is given an area of memory to store its value. While the variable is allocated, it can operate as a variable in the usual way to hold a value. A variable is deallocated when the system reclaims the memory from the variable, so it no longer has an area to store its value.

Local memory

The most common variables you use are "local" variables within functions such as the variables `num` and `result` in the following function. All of the local variables and parameters taken together are called its "local storage" or just its "locals", such as `num` and `result` in the following code:

```
int Square(int num) {
    int result;
    result = num * num;
    return result;
}
```

The variables are called "local" to capture the idea that their lifetime is tied to the function where they are declared. Whenever the function runs, its local variables are allocated. When the function exits, its locals are deallocated. For the above example, that means that when the `Square()` function is called, local storage is allocated for `num` and `result`. Statements like `result = num * num;` in the function use the local storage. When the function finally exits, its local storage is deallocated.

Here is a more detailed version of the rules of local storage...

1. When a function is called, memory is allocated for all of its locals. In other words, when the flow of control hits the starting '{' for the function, all of its locals are allocated memory. Parameters such as `num` and local variables such as `result` in the above example both count as locals. The only difference between parameters and local variables is that parameters start out with a value copied from the caller while local variables start with random initial values. This article mostly uses simple int variables for its examples, however local allocation works for any type: structs, arrays... these can all be allocated locally.
2. The memory for the locals continues to be allocated so long as the thread of control is within the owning function. Locals continue to exist even if the function temporarily

passes off the thread of control by calling another function. The locals exist undisturbed through all of this.

3. Finally, when the function finishes and exits, its locals are deallocated. This makes sense in a way — suppose the locals were somehow to continue to exist — how could the code even refer to them? The names like `num` and `result` only make sense within the body of `Square()` anyway. Once the flow of control leaves that body, there is no way to refer to the locals even if they were allocated. That locals are available ("scoped") only within their owning function is known as "lexical scoping" and pretty much all languages do it that way now.

Here is an example which shows how the simple rule "the locals are allocated when their function begins running and are deallocated when it exits" can build more complex behavior. The drawing shows the sequence of allocations and deallocations which result when the function `X()` calls the function `Y()`. The points in time `T1`, `T2`, etc. are marked in the code and the state of memory at that time is shown in the drawing.

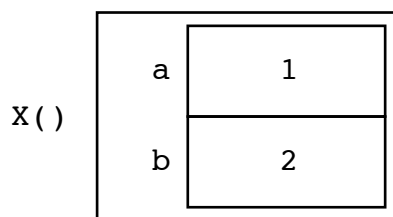
```
void X() {
    int a = 1;
    int b = 2;
    // T1

    Y(a);
    // T3
}

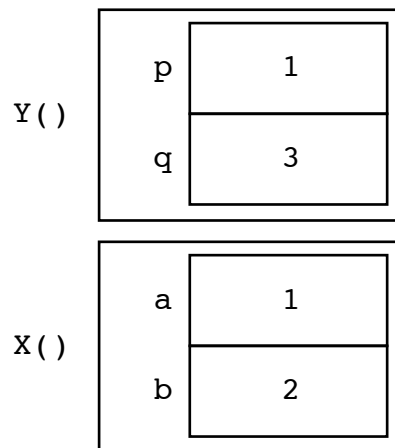
void Y(int p) {
    int q;
    q = p + 2;
    // T2
}
```

Drawing the sequence of the locals being allocated and deallocated:

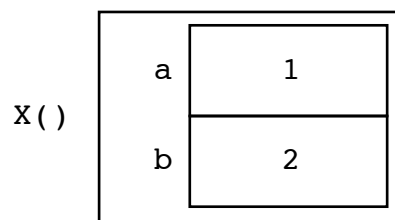
T1: `X()`'s locals have been allocated and given values



T2: $Y()$ is called with $p = 1$ and its locals are allocated. $X()$'s locals continue to be allocated



T3: $Y()$ exits and its local are deallocated. We are left only with $X()$'s locals.



Stack

In computer science, a stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list.

A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also means that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are typically those that have been in the list the longest. A common use of stacks at the Architecture level is as a means of allocating and accessing memory.

A call stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finishes executing. (The active subroutines are those which have been called but have not yet completed execution by returning.)

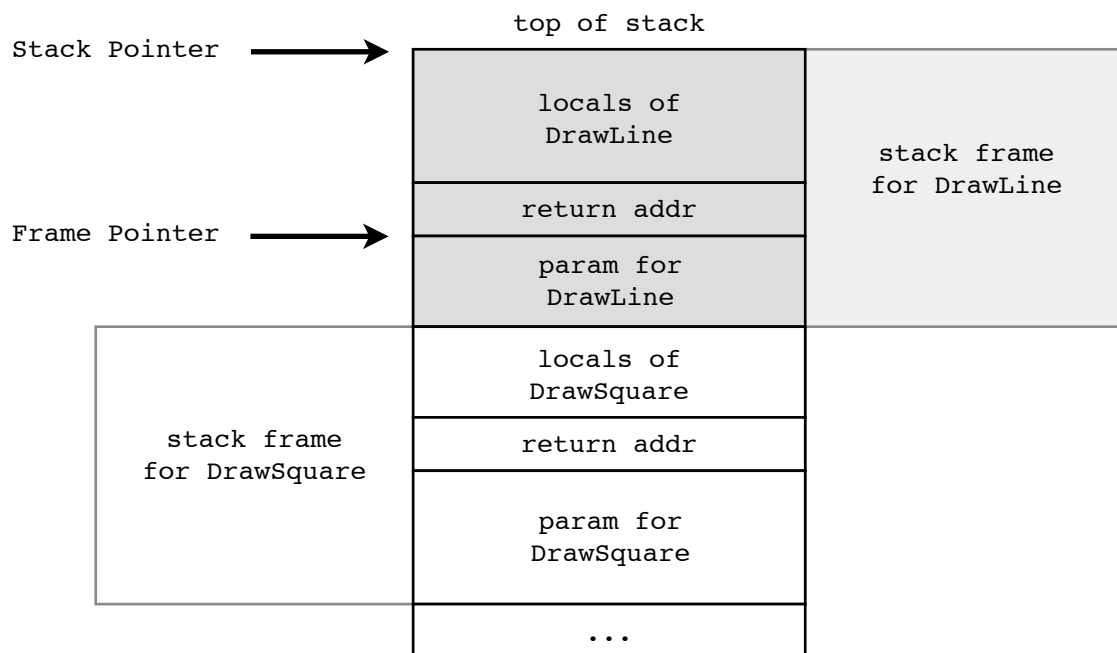
Since the call stack is organized as a stack, the caller pushes the return address onto the stack, and the called subroutine, when it finishes, pops the return address off the call stack

and transfers control to that address. If a called subroutine calls on to yet another subroutine, it will push another return address onto the call stack, and so on, with the information stacking up and unstacking as the program dictates. If the pushing consumes all of the space allocated for the call stack, an error called a stack overflow occurs, generally causing the program to crash. Adding a subroutine's entry to the call stack is sometimes called winding; conversely, removing entries is unwinding.

There is usually exactly one call stack associated with a running program (or more accurately, with each task or thread of a process).

Structure and use

A call stack is composed of stack frames (sometimes called activation records). These are machine dependent data structures containing subroutine state information. Each stack frame corresponds to a call to a subroutine which has not yet terminated with a return. For example, if a subroutine named DrawLine is currently running, having just been called by a subroutine DrawSquare, the top part of the call stack might be laid out like this (where the stack is growing towards the top):



The stack frame at the top of the stack is for the currently executing routine. In the most common approach the stack frame includes:

- ▶ space for the local variables of the routine
- ▶ the return address back to the routine's caller
- ▶ the parameter values passed into routine

The stack is often accessed via a register called the stack pointer, which also serves to indicate the current top of the stack. Alternatively, memory within the frame may be accessed

via a separate register, often termed the frame pointer, which typically points to some fixed point in the frame structure, such as the location for the return address.

In many systems a stack frame has a field to contain the previous value of the frame pointer register, the value it had while the caller was executing. For example, in the diagram above, the stack frame of DrawLine would have a memory location holding the frame pointer value that DrawSquare uses. The value is saved upon entry to the subroutine and restored for the return. Having such a field in a known location in the stack frame allows code to access each frame successively underneath the currently executing routine's frame.

Call site processing

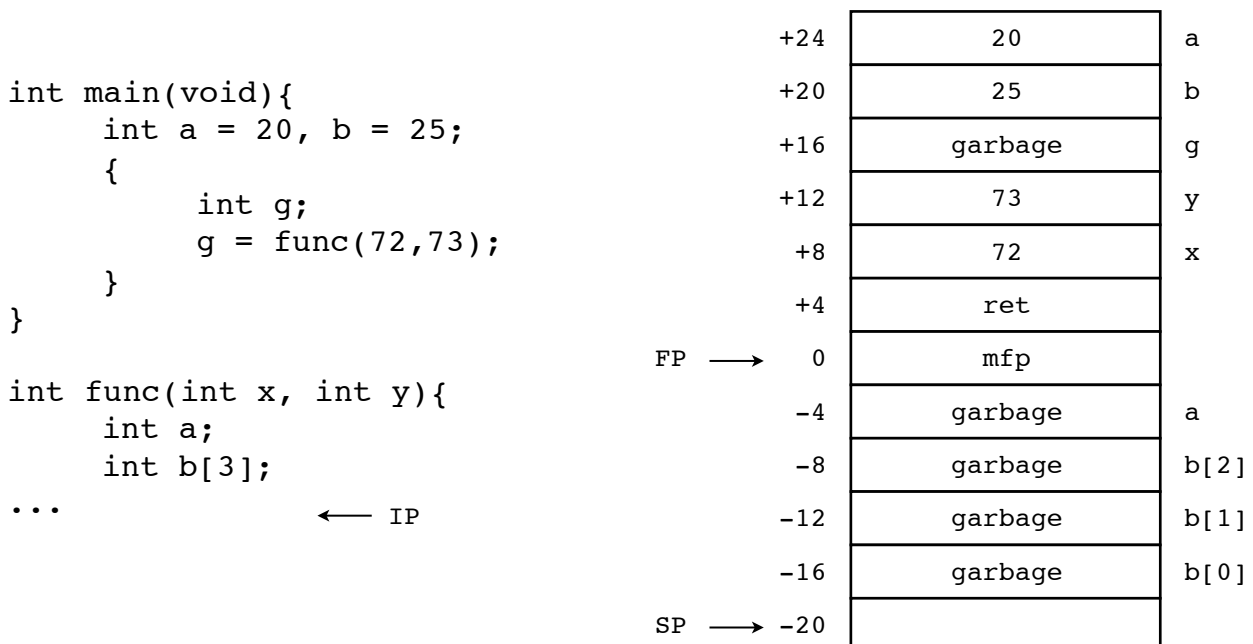
Usually the call stack manipulation needed at the site of a call to a subroutine is minimal (which is good since there can be many call sites for each subroutine to be called). The values for the actual arguments are evaluated at the call site, since they are specific to the particular call, and either pushed onto the stack or placed into registers, as determined by the calling convention being used. The actual call instruction, such as "Branch and Link," is then typically executed to transfer control to the code of the target subroutine.

Callee processing

In the called subroutine, the first code executed is usually termed the subroutine prologue, since it does the necessary housekeeping before the code for the statements of the routine is begun.

The prologue will commonly save the return address left in a register by the call instruction by pushing the value onto the call stack. Similarly, the current stack pointer and/or frame pointer values may be pushed. Alternatively, some instruction set architectures automatically provide comparable functionality as part of the action of the call instruction itself, and in such an environment the prologue need not do this.

If frame pointers are being used, the prologue will typically set the new value of the frame pointer register from the stack pointer. Space on the stack for local variables can then be allocated by incrementally changing the stack pointer.

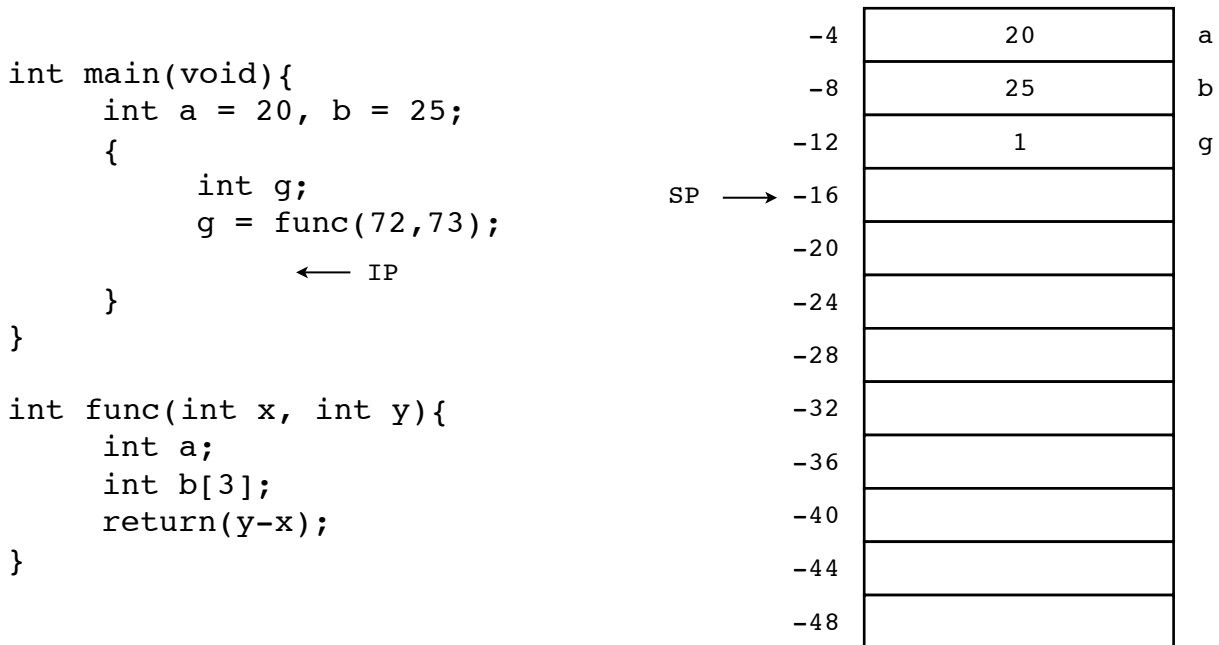


Return processing

When a subroutine is ready to return, it executes an epilogue that undoes the steps of the prologue. This will typically restore saved register values (such as the frame pointer value) from the stack frame, pop the entire stack frame off the stack by changing the stack pointer value, and finally branch to the instruction at the return address. Under many calling conventions the items popped off the stack by the epilogue include the original argument values, in which case there usually are no further stack manipulations that need to be done by the caller. With some calling conventions, however, it is the caller's responsibility to remove the arguments from the stack after the return.

Unwinding

Returning from the called function will pop the top frame off of the stack, perhaps leaving a return value.



Further readings

- <http://cslibrary.stanford.edu/102/PointersAndMemory.pdf>
- <http://www.eventhelix.com/RealtimeMantra/Basics/CToAssemblyTranslation.htm>

Heap

The memory management on the PowerPC can be used to frighten small children.

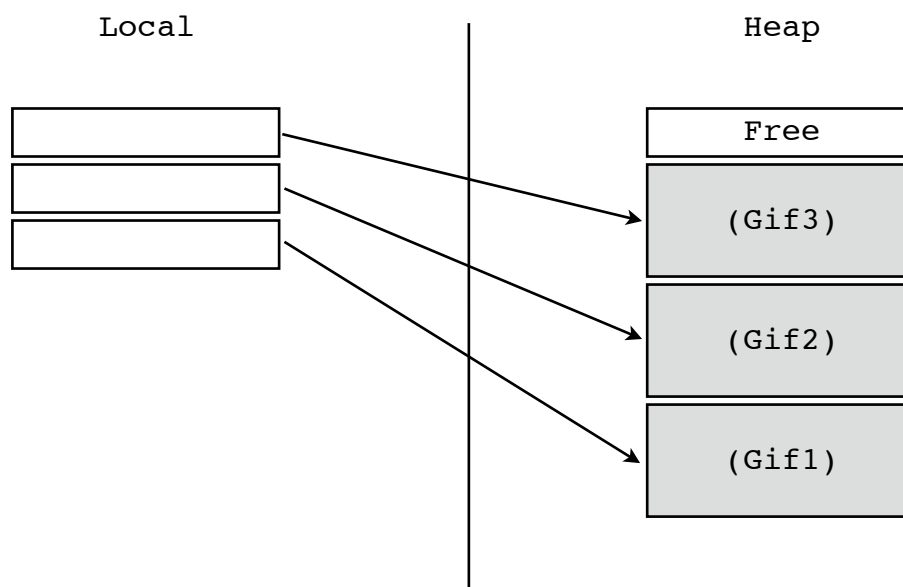
— Linus Torvalds

"Heap" memory, also known as "dynamic" memory, is an alternative to local stack memory. Local memory (Section 2) is quite automatic — it is allocated automatically on function call and it is deallocated automatically when a function exits. Heap memory is different in every way. The programmer explicitly requests the allocation of a memory "block" of a particular size, and the block continues to be allocated until the programmer explicitly requests that it be deallocated. Nothing happens automatically. So the programmer has much greater control of memory, but with greater responsibility since the memory must now be actively managed. The advantages of heap memory are:

1. Lifetime. Because the programmer now controls exactly when memory is allocated and deallocated, it is possible to build a data structure in memory, and return that data structure to the caller. This was never possible with local memory which was automatically deallocated when the function exited.
2. Size. The size of allocated memory can be controlled with more detail. For example, a string buffer can be allocated at run-time which is exactly the right size to hold a particular string. With local memory, the code is more likely to declare a buffer size 1000 and hope for the best.

Allocation

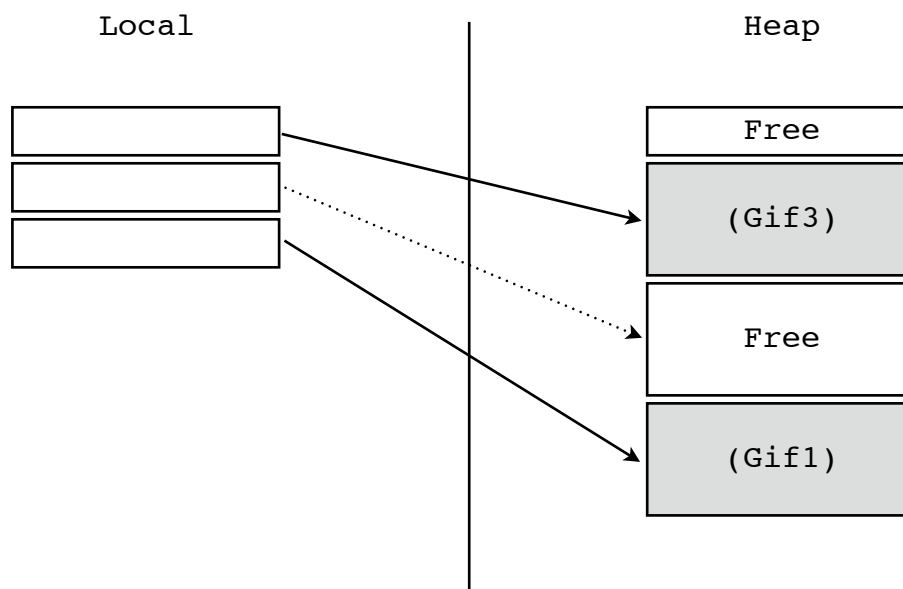
The heap is a large area of memory available for use by the program. The program can request areas, or "blocks", of memory for its use within the heap. In order to allocate a block of some size, the program makes an explicit request by calling the heap allocation function. The allocation function reserves a block of memory of the requested size in the heap and returns a pointer to it.



This program makes three allocation requests to allocate memory to hold three separate GIF images in the heap each of which takes 1024 bytes of memory. Each allocation request reserves a contiguous area of the requested size in the heap and returns a pointer to that new block to the program. Since each block is always referred to by a pointer, the block always plays the role of a "pointee" (Section 1) and the program always manipulates its heap blocks through pointers.

Deallocation

When the program is finished using a block of memory, it makes an explicit deallocation request to indicate to the heap manager that the program is now finished with that block. The heap manager updates its private data structures to show that the area of memory occupied by the block is free again and so may be re-used to satisfy future allocation requests. Here's what the heap would look like if the program deallocates the second of the three blocks:



After the deallocation, the pointer continues to point to the now deallocated block. The program must not access the deallocated pointee. This is why the pointer is drawn in gray — the pointer is there, but it must not be used. Sometimes the code will set the pointer to `NULL` immediately after the deallocation to make explicit the fact that it is no longer valid.

C specifics

In the C language, the library functions which make heap requests are `malloc()` ("memory allocate") and `free()`. The prototypes for these functions are in the header file `<stdlib.h>`. Although the syntax varies between languages, the roles of `malloc()` and `free()` are nearly identical in all languages:

- `void* malloc(unsigned long size);` The `malloc()` function takes an unsigned integer which is the requested size of the block measured in bytes. `malloc()` returns a pointer to a new heap block if the allocation is successful, and `NULL` if

the request cannot be satisfied because the heap is full. The C operator `sizeof()` is a convenient way to compute the size in bytes of a type — `sizeof(int)` for an `int` pointer, `sizeof(struct fraction)` for a `struct fraction` pointer.

- `void free(void* heapBlockPointer);` The `free()` function takes a pointer to a heap block and returns it to the free pool for later re- use. The pointer passed to `free()` must be exactly the pointer returned earlier by `malloc()`, not just a pointer to somewhere in the block. Calling `free()` with the wrong sort of pointer is famous for the particularly ugly sort of crashing which it causes. The call to `free()` does not need to give the size of the heap block — the heap manager will have noted the size in its private data structures. The call to `free()` just needs to identify which block to deallocate by its pointer. If a program correctly deallocates all of the memory it allocates, then every call to `malloc()` will later be matched by exactly one call to `free()`.

Further readings

- <http://www.exforsys.com/tutorials/c-language/dynamic-memory-allocation-in-c.html>
- <http://www.cs.cf.ac.uk/Dave/C/node11.html>
- http://www.cs.utah.edu/dept/old/texinfo/glibc-manual-0.02/library_3.html

GDB

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

— Brian W. Kernighan

Not every executable binary can be simply loaded into `gdb` for debugging. This is because it does not contain debugging symbols. These symbols more or less tells `gdb` where to look in the source when it's running a program.

Two things need to happen for `gdb` to be able to debug code. Firstly, you need to have the source tree in the same state as it was when you compiled your program. So, for example, if all the source files were compiled in `/home/user/program/src/`, and the binary was compiled in `/home/user/program/ui/`, then you can't move the sources elsewhere and expect `gdb` to “magically” know this.

The second thing is that you need to tell `gcc` to “implant” the symbols into the binary. You do this by specifying the `ggdb3` flag during compilation. So, for instance, compiling a `binary.c` to `binary`:

```
gcc -ggdb3 -o binary binary.c
```

It is worthwhile noting here that the `ggdb3` flag produces slower code. It should only be used when debugging.

Using gdb

Let's suppose we want to debug:

```
#include <stdio.h>

int func(int k)
{
    return(k + 1);
}

int main(void)
{
    int i;
    int k = 0;
    int j = 10;

    for(i = 0; i < 10; i++)
        k++;
    j = func(k);
    printf("%d\n", j);

    return(0);
}
```

The GNU Debugger can be started in many ways. The easiest way is:

```
gdb binary
```

This automatically loads the executable binary file `binary` and `gdb` is ready to take user input. When `gdb` is started, it should look somewhat like this:

```
GNU gdb 6.3.50-20050815 (Apple version gdb-1469) (Wed May  5
04:36:56 UTC 2010)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "x86_64-apple-darwin"...Reading symbols
for shared libraries .. done
```

```
(gdb)
```

Once you're in `gdb`, you can list the lines of your source-code, in order to navigate through your program. To do this, use the `list` command, or shorthand, `l`:

```
(gdb) list
2
3     int func(int k)
4     {
5         return(k + 1);
6     }
7
8     int main(void)
9     {
10         int i;
11         int k = 0;
(gdb)
12         int j = 10;
13
14         for(i = 0; i < 10; i++)
15             k++;
16         j = func(k);
17         printf("%d\n", j);
18
19         return(0);
20
21     }
(gdb)
```

Once a symbol-tipped binary is loaded into `gdb`, you need to explicitly tell `gdb` to execute it. You do so by the use of the command `run`. Here's an example:

```
(gdb) run
Starting program: /private/tmp/binary
Reading symbols for shared libraries +. done
11
```

```
Program exited normally.
(gdb)
```

You can also pass command line parameters by sitting them next to the run command. If you have already specified parameters, then a stand-alone run will automatically load your previously used command line arguments.

Breakpoints are probably the first thing you need to learn when using any debugging tool. When you define a breakpoint in some part of your program, gdb returns control back to you when it reaches that exact point. This is a very powerful feature, because you can stop the program at points where you suspect something might be happening.

Breakpoints are created using the break command. For instance, this is telling gdb to pass back control on line 13:

```
(gdb) b 13
Breakpoint 1 at 0x100000ec0: file test.c, line 13.
(gdb)
```

Once control is given back to you, the programmer, you can then use other commands to examine the state of your program. For instance, you can use print to examine the state of the variables currently in your scope.

```
(gdb) run
Starting program: /private/tmp/binary

Breakpoint 1, main () at test.c:14
14          for(i = 0; i < 10; i++)
(gdb)
```

```
(gdb) print j
$2 = 10
(gdb) print k
$3 = 0
(gdb) print i
$4 = 32767
(gdb)
```

Notice here that the number of the breakpoint is displayed. It's extremely important to note that each breakpoint has a unique number, because some commands we discuss later need to use that number to address that breakpoint.

You may also require to remove breakpoints from your debugging session. This can be done on per-breakpoint basis, or all at once via the command delete or del.

Calling delete with a numerical argument, causes the breakpoint associated with that numerical argument to be cleared. All breakpoints can be cleared by issuing delete without any arguments.

When you've been given control by gdb after it has reached a breakpoint, there are two commands that allow you to navigate through the sources. These are particularly useful for debugging iterative routines, and nested function calls.

The first of these commands is next or n. Next should be used when you simply want to remain in the scope of the current function. So, if you've reached a function call, using next will cause that function to be executed before control is returned to you. Example:

```
(gdb) n
15                               k++;
(gdb) print i
$5 = 0
```

The second command is step or s. Using step, allows you to get more into the detail of the program. It basically steps into a function call. For instance:

```
(gdb) b 16
Breakpoint 2 at 0x100000ed5: file test.c, line 16.
(gdb) continue
Continuing.

Breakpoint 2, main () at test.c:16
16          j = func(k);
(gdb) step
func (k=10) at test.c:5
5          return(k + 1);
(gdb)
```

The call stack is where we find the stack frames that control program flow. When a function is called, it creates a stack frame that tells the computer how to return control to its caller after it has finished executing. Stack frames are also where local variables and function arguments are 'stored'. We can look at these stack frames to determine how our program is running. Finding the list of stack frames below the current frame is called a backtrace.

Use the gdb command backtrace. In the backtrace below, we can see that we are currently inside func(), which was called from main():

```
(gdb) backtrace
#0 func (k=10) at test.c:5
#1 0x0000000100000edd in main () at test.c:16
(gdb)
```

Use the gdb command frame. Notice in the backtrace above that each frame has a number beside it. Pass the number of the frame you want as an argument to the command.

```
(gdb) frame 0
#0 func (k=10) at test.c:5
5          return(k + 1);
(gdb)
```

To look at the contents of the current frame, there are 3 useful gdb commands. info frame displays information about the current stack frame. info locals displays the list of local

variables and their values for the current stack frame, and info args displays the list of arguments.

```
(gdb) info frame
Stack level 0, frame at 0x7fff5fbff9f0:
  rip = 0x100000ea3 in func (test.c:5); saved rip 0x100000edd
  called by frame at 0x7fff5fbffa10
  source language c.
  Arglist at 0x7fff5fbff9e8, args: k=10
  Locals at 0x7fff5fbff9e8, Previous frame's sp is 0x7fff5fbff9f0
  Saved registers:
    rdi at 0x7fff5fbff9dc, rbp at 0x7fff5fbff9e0, rip at
0x7fff5fbff9e8
(gdb) info locals
No locals.
(gdb) info args
k = 10
(gdb)
```

For main():

```
(gdb) frame 1
#1 0x0000000100000edd in main () at test.c:16
16          j = func(k);
(gdb) info frame
Stack level 1, frame at 0x7fff5fbffa10:
  rip = 0x100000edd in main (test.c:16); saved rip 0x100000e94
  caller of frame at 0x7fff5fbff9f0
  source language c.
  Arglist at 0x7fff5fbffa08, args:
  Locals at 0x7fff5fbffa08, Previous frame's sp is 0x7fff5fbffa10
  Saved registers:
    rbp at 0x7fff5fbffa00, rip at 0x7fff5fbffa08
(gdb) info locals
i = 10
k = 10
j = 10
(gdb) info args
No arguments.
(gdb)
```

Further readings

- <http://www.gnu.org/software/gdb/documentation/>
- http://www.delorie.com/gnu/docs/gdb/gdb_toc.html

Memory segmentation

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

— Rich Cook

In a computer system using segmentation, an instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment. A segment has a set of permissions, and a length, associated with it. If the currently running process is allowed by the permissions to make the type of reference to memory that it is attempting to make, and the offset within the segment is within the range specified by the length of the segment, the reference is permitted; otherwise, a hardware exception is raised. When a program is executed it is read into memory where it resides until termination. The code allocates a number of special purpose memory blocks for different data types. A very common scheme, but not the only one, is depicted in the following table.

<p style="text-align: center;">STACK</p> <p>a very dynamic kind of memory located at it's top (high addresses) and growing downwards</p>
<p style="text-align: center;">memory not allocated yet</p> <p>Memory that will soon become allocated by the stack, that grows down. Stack will grow until it hits the administrative limit (predefined).</p>
<p style="text-align: center;">shared libraries</p>
<p style="text-align: center;">memory not allocated yet</p> <p>Memory that will soon become allocated by the heap growing up from underneath.</p>
<p style="text-align: center;">HEAP</p> <p>It is said that this is the most dynamic part of memory. It is dynamically allocated and freed in big chunks. The allocation process is rather complex (stub/buddy system) and is more time consuming than putting things on stack.</p>
<p style="text-align: center;">BSS</p> <p>Memory containing global variables of known (predeclared) size.</p>
<p style="text-align: center;">Constant data</p> <p>All constants used in a program.</p>
<p style="text-align: center;">Static program code</p>
<p style="text-align: center;">Reserved / other stuff</p>

Segments

In the PC architecture there are four basic read-write memory regions in a program: Stack, Data, BSS, and Heap.

- ▶ **DATA.** The Data area contains global and static variables used by the program that are initialized. This segment can be further classified into initialized read-only area and initialized read-write area. For instance the string defined by `char s[] = "hello world"`; in C and a C statement like `int debug = 1`; outside the main would be stored in initialized read-write area. And a C statement like `char *string = "hello world"`; makes the string literal "hello world" to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.
- ▶ **BSS.** In C, statically-allocated variables without an explicit initializer are initialized to zero (for arithmetic types) or a null pointer (for pointer types). Implementations of C typically represent zero values and null pointer values using a bit pattern consisting solely of zero-valued bits (though this is not required by the C standard). Hence, the bss section typically includes all uninitialized variables declared at the file level (i.e., outside of any function) as well as uninitialized local variables declared with the static keyword. An implementation may also assign statically-allocated variables initialized with a value consisting solely of zero-valued bits to the bss section.
- ▶ **HEAP.** The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (although, note that the use of brk/sbrk and a single "heap area" is not required to fulfil the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process' virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.
- ▶ **STACK.** The stack is a LIFO structure, typically located in the higher parts of memory. It usually "grows down" with every register, immediate value or stack frame being added to it. A stack frame consists at minimum of a return address.

And finally:

- ▶ **CODE.** In computing, a code segment, also known as a text segment or simply as text, is a phrase used to refer to a portion of memory or of an object file that contains executable instructions. It has a fixed size and is usually read-only. If the text section is not read-only, then the particular architecture allows self-modifying code. Read-only code is reentrant if it can be executed by more than one process at the same time. As a memory region, a code segment resides in the lower parts of memory or at its very bottom, in order to prevent heap and stack overflow from overwriting it.

Further readings

- <http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation>
- http://www.linuxforums.org/articles/understanding-elf-using-readelf-and-objdump_125.html

Process vs Thread

Threads [and] signals [are] a platform-dependent trail of misery, despair, horror and madness.

— Anthony Baxter

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

A computer program is a passive collection of instructions, a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

In general, a computer system process consists of (or is said to 'own') the following resources:

- ▶ An image of the executable machine code associated with a program.
- ▶ Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time.
- ▶ Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- ▶ Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- ▶ Processor state (context), such as the content of registers, physical memory addressing, etc. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

In computing, when a process forks, it creates a copy of itself. More generally, a fork in a multithreading environment means that a thread of execution is duplicated, creating a child thread from the parent thread.

Under Unix and Unix-like operating systems, the parent and the child processes can tell each other apart by examining the return value of the `fork()` system call. In the child process, the return value of `fork()` is 0, whereas the return value in the parent process is the PID of the newly-created child process.

The fork operation creates a separate address space for the child. The child process has an exact copy of all the memory segments of the parent process, though if copy-on-write semantics are implemented actual physical memory may not be assigned (i.e., both

processes may share the same physical memory segments for a while). Both the parent and child processes possess the same code segments, but execute independently of each other. When a `fork()` system call is issued, a copy of all the pages corresponding to the parent process is created, loaded into a separate memory location by the OS for the child process. But this is not needed in certain cases. Consider the case when a child executes an "exec" system call (which is used to execute any executable file from within a C program) or exits very soon after the `fork()`. When the child is needed just to execute a command for the parent process, there is no need for copying the parent process' pages, since exec replaces the address space of the process which invoked it with the command to be executed. In some cases, a technique called copy-on-write (COW) is used. With this technique, when a fork occurs, the parent process's pages are not copied for the child process. Instead, the pages are shared between the child and the parent process. Whenever a process (parent or child) modifies a page, a separate copy of that particular page alone is made for that process (parent or child) which performed the modification. This process will then use the newly copied page rather than the shared one in all future references. The other process (the one which did not modify the shared page) continues to use the shared version of the page. This technique is called copy-on-write since the page is copied when some process writes to it.

```
#include <stdio.h>    /* printf, stderr, fprintf */
#include <unistd.h>   /* _exit, fork */
#include <stdlib.h>   /* exit */
#include <errno.h>    /* errno */

int main(void)
{
    pid_t  pid;

    /* Output from both the child and the parent process
     * will be written to the standard output,
     * as they both run at the same time.
     */
    pid = fork();
    if (pid == 0)
    {
        /* Child process:
         * When fork() returns 0, we are in
         * the child process.
         * Here we count up to ten, one each second.
         */
        int j;
        for (j = 0; j < 10; j++)
        {
            printf("child: %d\n", j);
            sleep(1);
        }
        _exit(0); /* Note that we do not use exit() */
    }
    else if (pid > 0)
```

```

{
    /* Parent process:
    * When fork() returns a positive
    * number, we are in the parent process
    * (the fork return value is the PID of
    * the newly-created child process).
    * Again we count up to ten.
    */
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("parent: %d\n", i);
        sleep(1);
    }
    exit(0);
}
else
{
    /* Error:
    * When fork() returns a negative number, an error happened
    * (for example, number of processes reached the limit).
    */
    fprintf(stderr, "can't fork, error %d\n", errno);
    exit(EXIT_FAILURE);
}
}

```

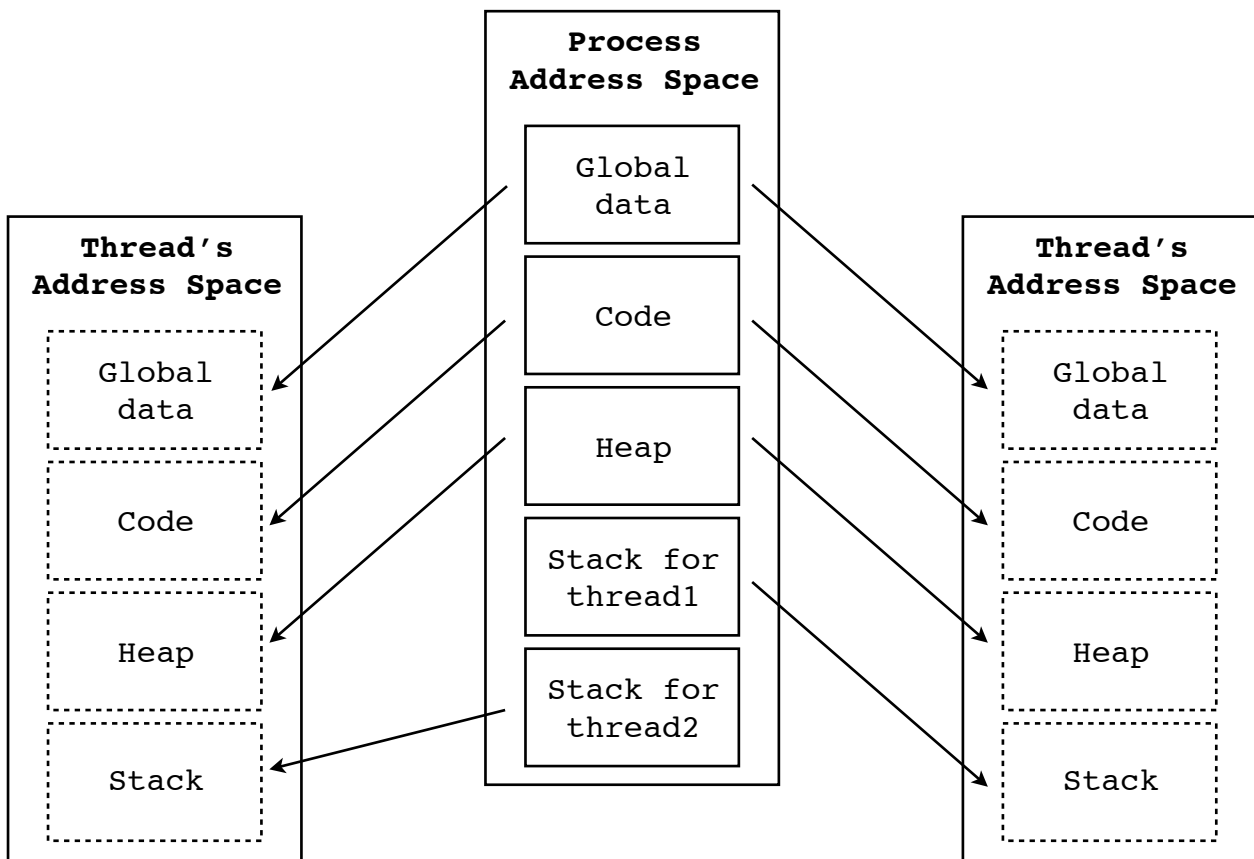
Threads

Threads differ from traditional multitasking operating system processes in that:

- ▶ processes are typically independent, while threads exist as subsets of a process
- ▶ processes carry considerable state information, whereas multiple threads within a process share state as well as memory and other resources
- ▶ processes have separate address spaces, whereas threads share their address space
- ▶ processes interact only through system-provided inter-process communication mechanisms.
- ▶ Context switching between threads in the same process is typically faster than context switching between processes.

Elements per process	Elements per thread
Address space	Program counter
Global variables	Registers
Open files	Stack

Elements per process	Elements per thread
Children processes	State
Pending alarms	
Signals and signal handlers	
Account informations	



Further readings

- <http://www.yolinux.com/TUTORIALS/ForkExecProcesses.html>
- <http://www.linux-tutorial.info/modules.php?name=MContent&pageid=83>
- <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>
- <http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/>
- <https://computing.llnl.gov/tutorials/pthreads/>

Linux shell

Dijkstra probably hates me

— Linus Torvalds, in *kernel/sched.c*

<code>cat</code>	Send a file to the screen in one go. Useful for piping to other programs
<code>gcc</code>	Compile a C program
<code>cd</code>	Change current directory
<code>chmod</code>	change file system modes of files and directories
<code>cp</code>	Copy file(s)
<code>date</code>	Shows current date
<code>file</code>	Tells you what sort of file it is
<code>grep</code>	Look for text in files. List out lines containing text
<code>gtar</code>	GNU version of the tar utility. Store directories and files together into a single archive file
<code>gzip</code>	GNU Compress files into a smaller space, or decompress .Z or .gz files.
<code>kill</code>	Kill, pause or continue a process. Can also be used for killing daemons.
<code>logout</code>	Closes the current shell. Also try "exit".
<code>ls</code>	Show lists of files or information on the files
<code>man</code>	Get instructions for a particular Unix command or a bit of Unix. Use space to get next page and q to exit.
<code>mkdir</code>	Create a new directory
<code>more</code>	Show a file one screen at a time
<code>mv</code>	Move file(s) or rename a file
<code>ps</code>	List processes on system
<code>pwd</code>	Show current working directory
<code>rm</code>	Delete (remove) files

<code>top</code>	Interactively show you the "top" processes on a system - the ones consuming the most computing (CPU) time. Press the "q" key in top to exit. Press the "k" key to kill a particular process. Press "r" to renice a process
<code>vi</code>	screen-oriented text editor

Further readings

- <http://ss64.com/bash/>
- http://linuxcommand.org/learning_the_shell.php
- vic.gedris.org/Manual-ShellIntro/1.2/ShellIntro.pdf

Bash Programming

Unix is a junk OS designed by a committee of PhDs

— Dave Cutler

```
#!/bin/bash
echo Hello World
```

Redirection

```
ls -l > ls-l.txt
grep da * 2> grep-errors.txt
grep da * 1>&2
grep * 2>&1
rm -f $(find / -name core) &> /dev/null
```

Pipes

```
ls -l | sed -e "s/[aeio]/u/g"
ls -l | grep "\.txt$"
```

Variables

```
#!/bin/bash
STR="Hello World!"
echo $STR
```

```
#!/bin/bash
OF=/var/my-backup-$(date +%Y%m%d).tgz
tar -czf $OF /home/me/
```

Conditionals

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
    echo expression evaluated as true
else
    echo expression evaluated as false
fi
```

Loops for, while and until

```
#!/bin/bash
for i in $( ls ); do
    echo item: $i
done
```

```
#!/bin/bash
for i in `seq 1 10`; do
```

```
do
    echo $i
done
```

```
#!/bin/bash
COUNTER=0
while [ $COUNTER -lt 10 ]; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done
```

```
#!/bin/bash
COUNTER=20
until [ $COUNTER -lt 10 ]; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

Functions

```
#!/bin/bash
function quit {
    exit
}
function hello {
    echo Hello!
}
hello
quit
echo foo
```

```
#!/bin/bash
function quit {
    exit
}
function e {
    echo $1
}
e Hello
e World
quit
echo foo
```

Command line

```
#!/bin/bash
if [ -z "$1" ]; then
    echo usage: $0 directory
    exit
fi
SRCD=$1
TGTD="/var/backups/"
```

```
OF=home-$(date +%Y%m%d).tgz
tar -czf $TGTD$OF $SRCD
```

Reading user input

```
#!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

Arithmetic relational operators

```
-lt (<)
-gt (>)
-le (<=)
-ge (>=)
-eq (==)
-ne (!=)
```

Capturing a commands output

```
#!/bin/bash
DBS=`mysql -uroot -e"show databases" `
for b in $DBS ;
do
    mysql -uroot -e"show tables from $b"
done
```

Further readings

- <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>
- <http://tldp.org/LDP/abs/html/>