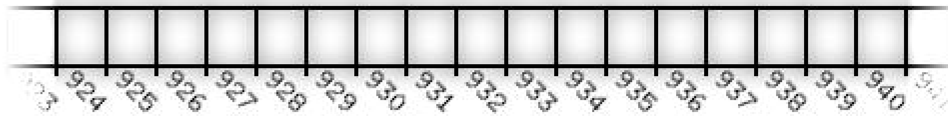
Pointers

Using Variables

Essentially, the computer's memory is made up of bytes. Each byte has a number, an address, associated with it. The picture below represents several bytes of a computer's memory. In the picture, addresses 924 thru 940 are shown.



Try:

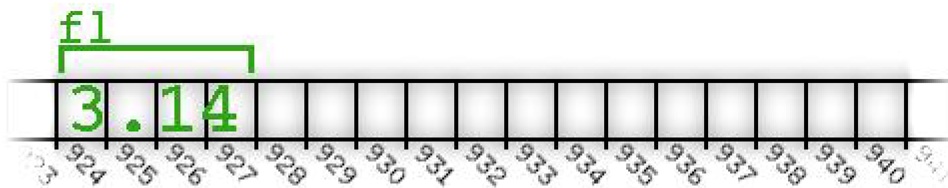
C++

```
1:#include <iostream>
2:main()
 3:{
 4:   float fl=3.14;
 5:   std::cout << fl << std::endl;
 6:}
```

C

```
1:#include <stdio.h>
2:main()
 3:{
 4:   float fl=3.14;
 5:   printf("%.2f\n", fl);
 6:}
```

At line (4) in the program above, the computer reserves memory for `fl`. In our examples, we'll assume that a `float` requires 4 bytes. Depending on the computer's architecture, a `float` may require 2, 4, 8 or some other number of bytes.



When `fl` is used in line (5), two distinct steps occur:

1. The program finds and grabs the address reserved for `fl`--in this example 924.
2. The contents stored at that address are retrieved

To generalize, whenever *any* variable is accessed, the above two distinct steps occur to retrieve the contents of the variable.

The illustration that shows 3.14 in the computer's memory can be

misleading. Looking at the diagram, it appears that "3" is stored in memory location `924`, "." is stored in memory location `925`, "1" in `926`, and "4" in `927`. Keep in mind that the computer actually uses an algorithm to convert the floating point number 3.14 into a set of ones and zeros. Each byte holds 8 ones or zeros. So, our 4 byte `float` is stored as 32 ones and zeros (8 per byte times 4 bytes). Regardless of whether the number is 3.14, or -273.15, the number is always stored in 4 bytes as a series of 32 ones and zeros.

Separating the Steps

Two operators are provided that, when used, cause these two steps to occur separately.

operator	meaning	example
`&`	do only step 1 on a variable	`&fl`
`*`	do step 2 on a number(address)	`*some_num`

Try this code to see what prints out:

C++

```
1:#include <iostream>
2:main()
3:{
4:  float fl=3.14;
5:  std::cout << "fl's address=" <<
(unsigned int) &fl << std::endl;
6:}
```

C

```
1:#include <stdio.h>
2:main()
3:{
4:  float fl=3.14;
5:  printf("fl's address=%u\n",
(unsigned int) &fl);
6:}
```

On line (5) of the example, The `&` operator is being used on `fl`. On line (5), only step 1 is being performed on a variable:

1. The program finds and grabs the address reserved for fl...

It is `fl`'s address that is printed to the screen. If the `&` operator had not been placed in front of `fl`, then step 2 would have occurred as well, and 3.14 would have been printed to the screen.

The `(unsigned int)` phrase will be discussed later. It is there so that `&addr` will print out as a non-negative number. It has been shown in gray to indicate that you must include it for the program to compile properly but that it is not relevant to this current discussion.

Keep in mind that an address is really just a simple number. In fact, we can store an address in an integer variable. Try this:

C++

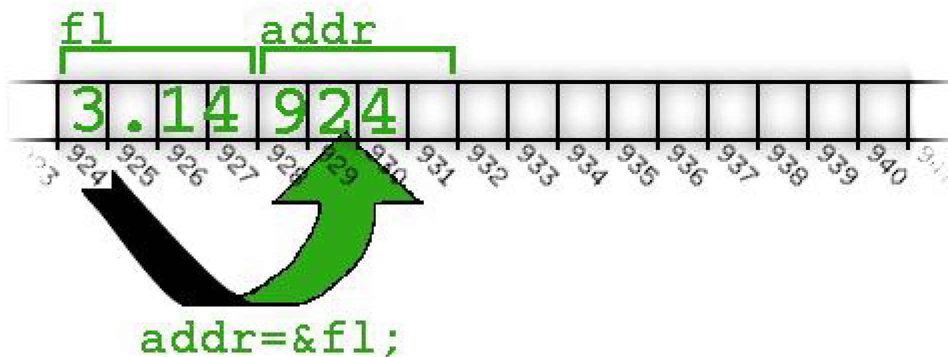
```
1:#include <iostream>
2:main()
```

C

```
1:#include <stdio.h>
2:main()
```

```
3:{
4:   float fl=3.14;
5:   unsigned int addr=(unsigned int)
&fl;
6:   std::cout << "fl's address=" <<
addr << std::endl;
7:}
```

```
3:{
4:   float fl=3.14;
5:   unsigned int addr=(unsigned
int) &fl;
6:   printf("fl's address=%u\n",
addr);
7:}
```



The above code shows that there is nothing magical about addresses. They are just simple numbers that can be stored in integer variables.

> The `unsigned` keyword at the start of line (5) simply means that the integer will not hold negative numbers. As before, the `(unsigned int)` phrase has been shown in gray. It must be included for the code to compile, but is not relevant to this discussion. It will be discussed later.

Now let's test the other operator, the `*` operator that retrieves the contents stored at an address:

C++

```
1:#include <iostream>
2:main()
3:{
4:   float fl=3.14;
5:   unsigned int addr=(unsigned int)
&fl;
6:   std::cout << "fl's address=" <<
addr << std::endl;
7:   std::cout << "addr's contents=" <<
* (float*) addr << std::endl;
8:}
```

C

```
1:#include <stdio.h>
2:main()
3:{
4:   float fl=3.14;
5:   unsigned int addr=(unsigned
int) &fl;
6:   printf("fl's address=%u\n",
addr);
7:   printf("addr's
contents=%.2f\n", * (float*)
addr);
8:}
```

In line (7), step 2 has been performed on a number:

2. The contents stored at that address [`addr`] are retrieved

In order to make line (7) work, a little "syntax sugar" had to be added for the program to compile. Like before, `(float*)` is shown in gray because it is not relevant to the current discussion. For the sake of this discussion, just read "`*(float*)addr`" as "`*addr`" (that is, ignore the stuff in gray). The code shown in gray will be discussed later.

OK, But why do we need & and *

We have shown that 2 distinct steps occur when accessing a variable, and that we can make those steps occur separately. But why is this useful?

To see why, let's first look at how functions work in C/C++. Try this code:

C++

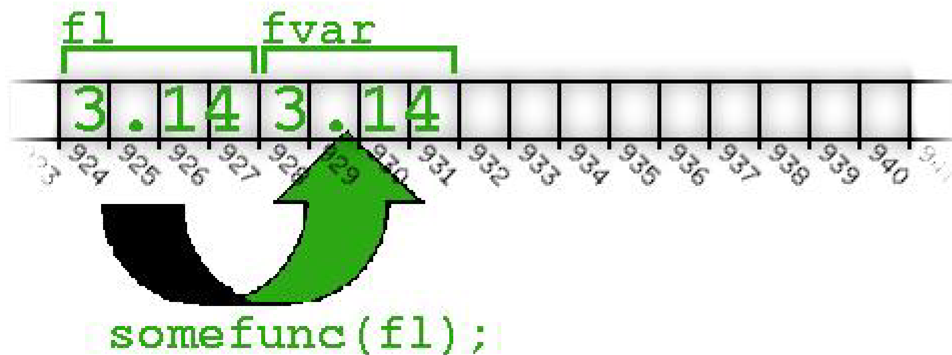
```
1:#include <iostream>
2:void somefunc(float fvar)
 3:{
 4:   fvar=99.9;
 5:}
 6:main()
 7:{
 8:   float fl=3.14;
 9:   somefunc(fl);
10:   std::cout << fl << std::endl;
11:}
```

C

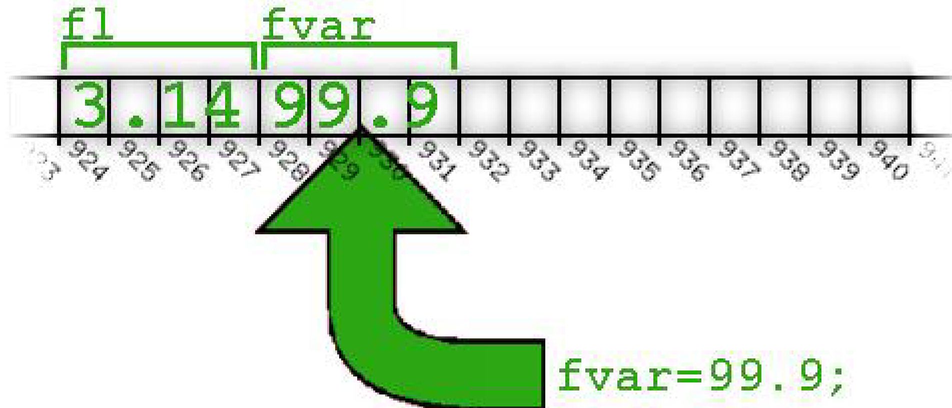
```
1:#include <stdio.h>
2:void somefunc(float fvar)
 3:{
 4:   fvar=99.9;
 5:}
 6:main()
 7:{
 8:   float fl=3.14;
 9:   somefunc(fl);
10:   printf("%.2f\n", fl);
11:}
```

What prints out? 3.14? 99.9? It turns out that 3.14 prints out. The general term used to describe this behavior is *pass by value*. When `somefunc(fl)` is called at line 9:

1. Execution jumps to line (2) to run the function
2. `fvar` is created as its own variable and `fl`'s value is copied into `fvar`



3. On line (4), 99.9 is assigned to fvar



4. Now that the function is finished, execution resumes in `main` where it left off (line 10). The `fl` variable is unchanged, 3.14 prints out.

We can circumvent this *pass by value* behavior and change values passed into functions by using the `&` and `*` operators.

C++

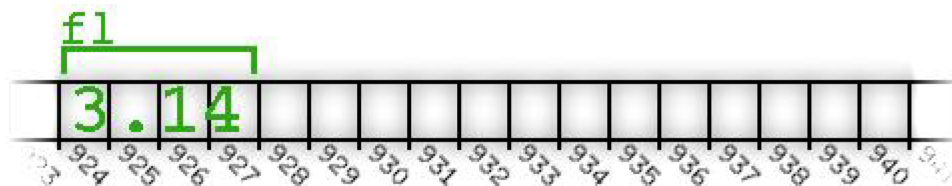
```
1:#include <iostream>
2:void somefunc(unsigned int fptr)
 3:{
 4:   *(float*)fptr=99.9;
 5:}
 6:
 7:main()
 8:{
 9:   float fl=3.14;
10:   unsigned int addr=(unsigned
int) &fl;
11:   somefunc(addr);
12:   std::cout << fl << std::endl;
13:}
```

C

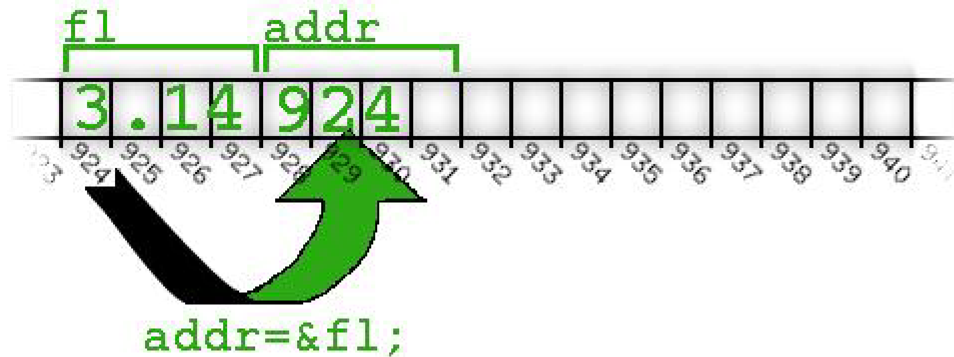
```
1:#include <stdio.h>
2:void somefunc(unsigned int fptr)
 3:{
 4:   *(float*)fptr=99.9;
 5:}
 6:
 7:main()
 8:{
 9:   float fl=3.14;
10:   unsigned int addr=(unsigned
int) &fl;
11:   somefunc(addr);
12:   printf("%.2f\n", fl);
13:}
```

Quite simply, the two steps that normally occur when accessing a variable are being separated to allow us to change the variable's value in a different function.

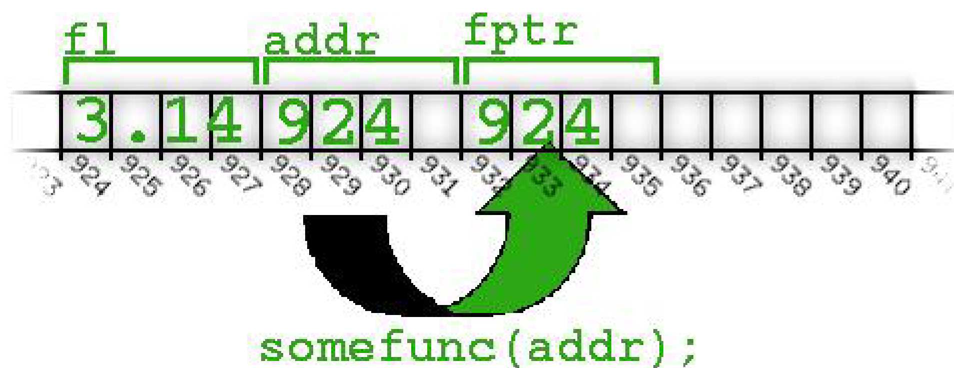
1. The floating point variable fl is created at line (9) and given the value 3.14



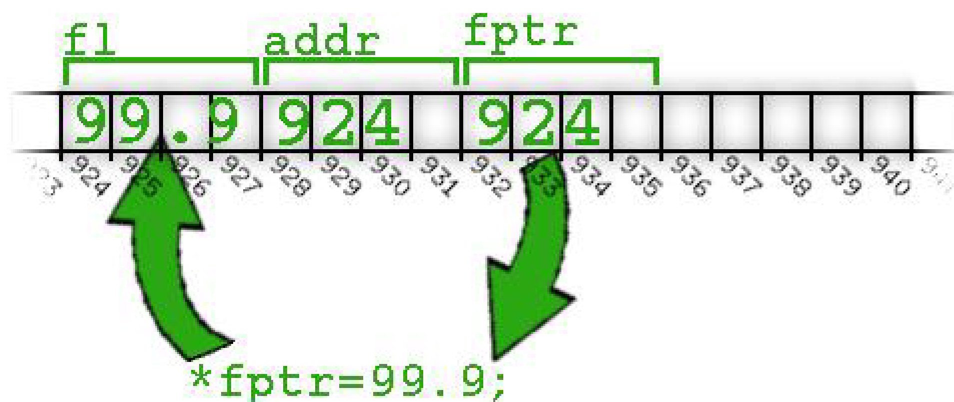
2. The `&` operator is used on fl at line (10) (do only step 1, get the address). The address is stored in the integer variable `addr`.



3. The function `somefunc` is called at line (11) and `fl`'s address is passed as an argument.
4. The function `somefunc` begins at line (2), `fptr` is created and `fl`'s address is copied into fptr.



5. The `*` operator is used on `fptr` at line (4) -- do step 2, the contents stored in an address are retrieved. In this example, the contents at address 924 are retrieved.
6. The contents at address 924 are assigned the value `99.9`.



7. The function finishes. Control returns to line (12).
8. The contents of `fl` are printed to the screen.

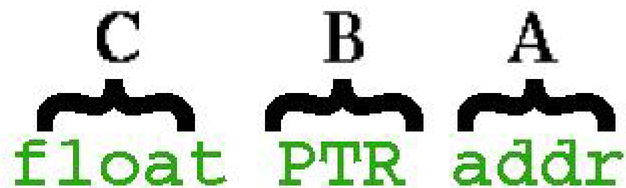
Pointer Variables

Even though we have shown that an address is nothing more than a simple integer, the creators of the language were afraid we might confuse variables in our programs. We might confuse integers we intend to use for program values (e.g. variables storing ages, measurements, counters, etc.) with integers we intend to use for holding the addresses of our variables.

The language creators decided the best way to eliminate confusion was to create a different *type* of variable for holding addresses. A first attempt at this might have looked something like this:

```
1:...
2:   float fl=3.14;
3:   float Ptr addr = &fl;
4:...
```

On line (3), here is how to describe the addr variable:



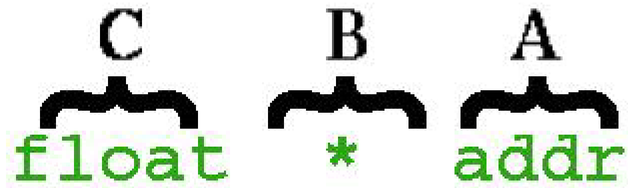
(A) `addr` is an integer. **(B)** However, it is a special integer designed to hold the address of a **(C)** `float`

In the code above, line (3) Is close to what the creators of the language wanted except for one thing: using `Ptr` would require introducing another keyword into the language. If there is one thing that all C instructors like to brag about, it is how there are only a very small number of keywords in the language. Well, using line (3) as shown above would mean adding `Ptr` as another keyword to the language.

To avoid this threat to the very fabric of the universe, the creators cast about for something already being used in the language that could do double duty as `Ptr` shown above. What they came up with was the following:

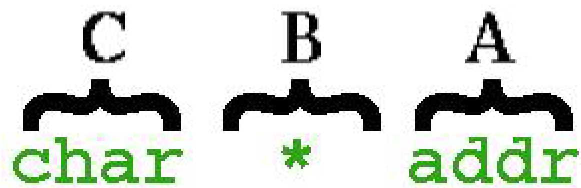
```
1:...
2:   float fl=3.14;
3:   float * addr = &fl; 4:...
```

Even with the `*` instead of `Ptr`, `addr` is described the same way:

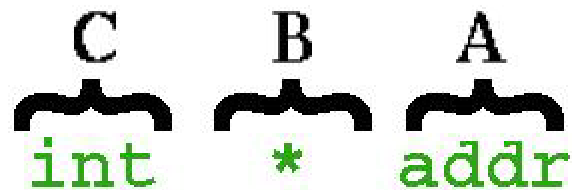


(A) `addr` is an integer. **(B)** However, it is a special integer designed to hold the address of a **(C)** `float`

These variables are described this way, regardless of the type:



(A) `addr` is an integer. **(B)** However, it is a special integer designed to hold the address of a **(C)** `char`



(A) `addr` is an integer. **(B)** However, it is a special integer designed to hold the address of an **(C)** `int`

This "...special integer..." way of describing these variables is a mouthful, so we shorten it and just say "addr is a float pointer" or "addr is a pointer to a float" (or char, or int, etc.).

Unfortunately, the language creators chose the `*` character to replace Ptr. The `*` character is confusing because the `*` character is also used to get the contents at an address ("do step 2 on a number"). **These two uses of the `*` character have nothing to do with each other.**

What is all that "syntax sugar" anyway? (Casting)

Let's take one last look at our original code that illustrates the utility of separating out steps 1 & 2.

C++ **C**

```
1:#include <iostream>
2:void somefunc(unsigned int fptr)
 3:{
 4:   *(float*)fptr=99.9;
 5:}
 6:
 7:main()
 8:{
 9:   float fl=3.14;
10:   unsigned int addr=(unsigned
int) &fl;
11:   somefunc(addr);
12:   std::cout << fl << std::endl;
13:}
```

```
1:#include <stdio.h>
2:void somefunc(unsigned int fptr)
 3:{
 4:   *(float*)fptr=99.9;
 5:}
 6:
 7:main()
 8:{
 9:   float fl=3.14;
10:   unsigned int addr=(unsigned
int) &fl;
11:   somefunc(addr);
12:   printf("%.2f\n", fl);
13:}
```

In nearly all of the code samples, you have been asked to ignore certain bits of the code. These bits of code have always appeared around those areas where we are either taking the address of a variable or getting the contents at an address (doing step 1 or step 2 on a variable)

Those bits of "syntax sugar" are there to keep the compiler from complaining. The first example of this in the above program is on line (10).

On line (10) we are taking the address of the floating point number `fl` ("do only step 1 on a number"). After we get that address, we store it in `addr`.

Why would the compiler complain? Because when we get assign the address of `fl` to `addr`, the compiler does not expect `addr` to be an `unsigned int`. The compiler expects `addr` to be a `float *`. That is, *a special integer designed to hold the address of a float*. To keep the compiler from complaining, we tell the compiler to treat `&fl` as an `unsigned int` rather than a `float *`.

This "syntax sugar" that causes the compiler to treat variables and expressions differently is called *casting*. The way a programmer describes line (10) is: "The address of `fl` is being cast into an `unsigned int` and assigned to `addr`"

The other place casting occurs is on line (4). On line (4), we are getting the contents at an address ("do step 2 on a number/address"). Why would the compiler complain? Because the compiler should get the contents of the address of a float. The address of our float is in stored in `fptr`, which is an `unsigned int`, not a `float *`. We tell the compiler to treat `fptr` as the address of a floating point number by casting it into a `float *`. Once we tell the compiler this, we can get the contents at the address without complaint.

Putting it all together

From the previous section, you might be left with the impression that whenever you deal with addresses and pointers, there is a lot of casting. Not so. The only reason our examples up till now have required casting is because we were storing our addresses in

`unsigned int` variables. The language designers want us to store addresses in the "special integer" variables, that is, the pointer variables they designed for just such a purpose.

Once we replace our `unsigned int` variables with these pointer variables, none of the casting is required:

C++

```
1:#include <iostream>
2:void somefunc(float* fptr)
 3:{
 4:   *fptr=99.9;
 5:}
 6:
 7:main()
 8:{
 9:   float fl=3.14;
10:   float* addr = &fl;
11:   somefunc(addr);
12:   std::cout << fl << std::endl;
13:}
```

C

```
1:#include <stdio.h>
2:void somefunc(float* fptr)
 3:{
 4:   *fptr=99.9;
 5:}
 6:
 7:main()
 8:{
 9:   float fl=3.14;
10:   float* addr = &fl;
11:   somefunc(addr);
12:   printf("%.2f\n", fl);
13:}
```

- On line (10), when we take the address of `fl` the address is assigned to a variable designed to hold it. No casting is required.
- When `addr` is passed to the function in line (11), `addr` is copied to `fptr` on line (2).
- Line (2) shows that `fptr` is created as a float pointer, that is a variable designed to hold the address of a floating point number. As a result, no casting is needed on line (4) where the contents at the address are retrieved.

Revision History

2002 June 02	Updated the C++ I/O preprocessor directives and I/O calls to conform to standard.
2001 April 30	Some minor corrections.
1999 March 19	Added C version of code. Minor corrections to text.

Miscellaneous

The graphics in this tutorial were created using the freely distributed image manipulation program The GIMP. Information on The GIMP can be found at http://www.gimp.org/

Please contact me with any errata, comments, suggested changes, or improvements:
tgibson@augustcouncil.com

The code in this tutorial that stores addresses in `unsigned int`'s may fail on a very few compilers, particulary older compilers. If this is the case with your compiler, try using `unsigned long` instead of `unsigned int`.

Copyright 2001-2002 Todd A. Gibson. All Rights Reserved.

While this document is copyright by me with all rights reserved, permission is granted to freely distribute verbatim copies of this document provided that no modifications outside of formatting be made, and that this notice remain intact.

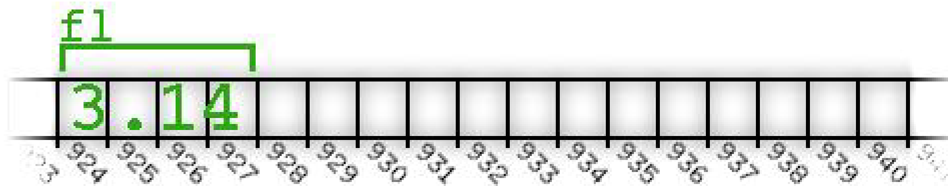
Computer Addresses

For the sake of this tutorial, drawings of memory are shown in the form of a ribbon divided into boxes. Each box represents a single byte with its own address. The sample addresses shown in the drawings run from 924 to 940.

Many of the sample programs in this tutorial will print addresses to the screen. Typically, the address values shown when running these programs will be quite high. It isn't unusual to see something like 3221223612 displayed for an address.

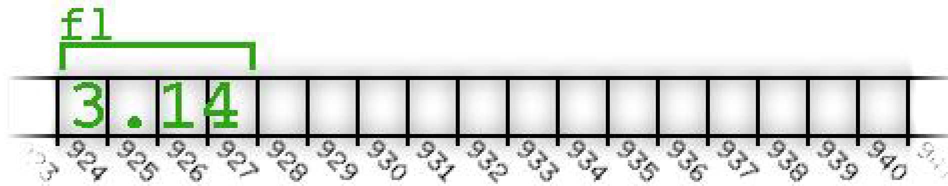
Don't worry that the addresses you see when running the sample programs are much larger than those shown in the drawings. Small 3 digit addresses are used in the drawings because it is much easier on the eyes than trying to read 10 digit addresses underneath each byte of memory.

1. ***The program finds and* grabs the address *reserved for `f1`-- in this example 924***



In our example, `f1` is not a single byte. It is made up of 4 bytes (with 4 addresses) In step 1, When the program "grabs the address", it grabs only the address of `f1`'s first byte--924
.

2. *The contents stored at that address are* retrieved



The program knows `fl` is a float, and that all floats have four bytes. Therefore, when the program does step 2, it knows to retrieve the contents in the 4 bytes starting at address 924.

The reason for pointer variables

Eliminating confusion is not an entirely accurate reason for pointer variables. A primary motivation for having a pointer type for each variable type is to help the compiler. Referring back to an earlier example, when we get the contents at an address ("do step 2 on a number"), the compiler must know *how* to get the contents at the address.

The compiler needs to know two things when getting the contents at an address:

- How many bytes starting at the given address make up the value. For example, does the address refer to a 2 byte `short int` or a 4 byte `float` or an 8 byte `double?`
- How to convert those bytes into the value. That is, the process for converting 4 bytes into a `float` is different than the process for converting 2 bytes into a `short int`.

So "eliminating confusion" ends up being more of a side-effect of having pointers rather than the reason for having them. That being said, there are programming languages that get by without typing pointers. For example, in the FORTH programming language, an address could just as easily point to a subroutine as point to a floating point number. It's up to the programmer to use the pointer correctly.

Pointer declarations and expressions

It is only a half-truth that these two uses of the `*` character have nothing to do with each other. It is well documented that the `*` character was chosen for these different uses because both uses involve pointers and therefore merit similar syntax.

The important point to come away with is that using the `*` character in *declarations*:

```
float *fptr;
```

is different from using it in *expressions*:

```
float fl;
...
fl=82.3 + *fptr;
```

From this perspective, the two uses of the `*` character have nothing to do with each other.

Casting

When a variable is *cast* into a different type, the actual variable is not harmed. Here is a snippet from the program currently being examined in the tutorial:

```
 2:void somefunc(unsigned int fptr)
 3:{
 4:   *(float*)fptr=99.9;
 5:}
```

Try adding this statement between lines (4) and (5):

```
  std::cout << fptr << std::endl;
```

This line just prints `fptr` to the screen. But the point is, that `fptr` is printed to the screen *as an* *`unsigned int`*. Even though `ftpr` was cast into a `float*` in line 4, `fptr` was not permanently transformed into a `float*`.

Casting merely changes the way the compiler *uses* the variable at the point where the cast is made.