C++ Programming

Dr. Yihsiang Liow   (December 21, 2025)

Contents

26. Pointers

OBJECTIVES

- Declare pointers
- Assign values to pointers, address-of operator, and NULL
- Comparison of address values
- Use pointers as function parameters modify values
- Use pointers as function parameters speedup function call
- Returning references and pointers
- Dynamic memory manage for non-array values
- Memory deallocation and preventing memory leaks

Now we come to a concept that is extremely useful and powerful in CS.

Drumroll . . . *pointers!!!*

Pointers are variables that can hold physical memory addresses. By pointing a pointer to any memory location in your computer, you can access the data at that point in your computer's memory. Both data and code are stored in your computer memory. Therefore with pointers you can access any data and code. This has several consequences.

- Pointers allow you to write a function that allows pass-by- reference. In fact your references should really be thought of as pointers under the hood.
- In terms of data, pointers allow you to access a part of your program's memory called the free store or memory heap. This allows you to control memory usage and have better control over memory management, requesting memory only when you need it and releasing it back to the system when it's not needed.
- By embedding pointer values into data, we can create very complex relations between data values by linking them together. This allows you to build data structures to model trees, graphs, etc. Graphs are probably one of the most important mathematical structures in CS. A road network is basically a graph. A computer network is basically a graph. Social graph that describes person-to-person relationship in social media is basically a graph. Etc., etc., etc., etc., etc. . . . !!!
- Since code also lives in your computer's memory, pointers can also point to code such as functions. With pointers, you can

actually pass functions (technically speaking the memory location of functions) into functions, i.e., functions can become arguments.

- Nowadays, hardware devices are usually "mapped" to memory locations too. That means that pointers can also be used to access devices to perform I/O.
- Etc!!! ... not to mention bizarre things like code that rewrites itself.

All the reasons have to do with using pointers to achieve some computational goals. Another really important reason why pointers are important is this: Pointers are not just some abstract idea of computations in the theoretical sense. Pointers are fundamental to actual computer architecture in the sense that pointers or memory addresses are understood by your hardware. For many programming languages, you cannot access pointers or memory addresses directly. Those languages will hide pointers away from programmers by providing some language feature(s) to achieve the same goal, bypassing pointers and memory addresses. This is no big deal if all you want to do is to achieve the high level computational goals. However, if you do need to dive into the guts of your computer, you will have to know pointers very well. This also means that people who are trained only in a language that does not provide means to access pointer or memory addresses will have problems doing low-level programming or will have problems fully understanding areas such as assembly language programming and computer architecture. And low level programming does happen in the real world for instance in the case of systems programming and game development.

Because the landscape of pointers is huge let me tell you what's the plan

- First I will talk about general concepts related to pointers: memory addresses, pointer values, pointer operations, etc. Another basic concept you need to know is pointer arithmetic. But I'll delay pointer arithmetic till the second set of notes on pointers so that we can go into some applications of pointers quickly
- For the first application of pointers, we'll see that pointers allow us to directly access and manipulation data from anywhere. In particular, a function can access data that is in another function. You recall that this is exactly what references are used for. In fact your C++ compiler actually converts references to pointers.

So understanding pointers is absolutely crucial to understanding references. And if you know pointers well, you don't even need references. In fact in the original C language – the ancestor of C++ – there is no concept of references.

- The second application is similar to the first: we'll call functions with pointers (we'll also look at passing in reference) and possibly returning pointers (we'll also look at returning reference). In this case, the goal is to speed up function call.
- For the third application, we'll go into an area in your computer called the heap and perform dynamic memory management ourselves. (The heap is also called the free store.) This means that we can ask the heap to give us an *int* value when we need it and give the *int* value back to the heap when we're done with it. This is very different from using an *int* value that belongs to an *int* variable. For an *int* variable, the value exists when you declare the *int* variable. The value is given back to the computer when the variable goes out of scope. You don't have as much control over memory usage for your regular variables. To use the heap ... you must use pointers.

In the next set of notes on pointers, I'll talk about pointer arithmetic, the relationship between pointers and arrays, and dynamic memory management of (dynamic) arrays.
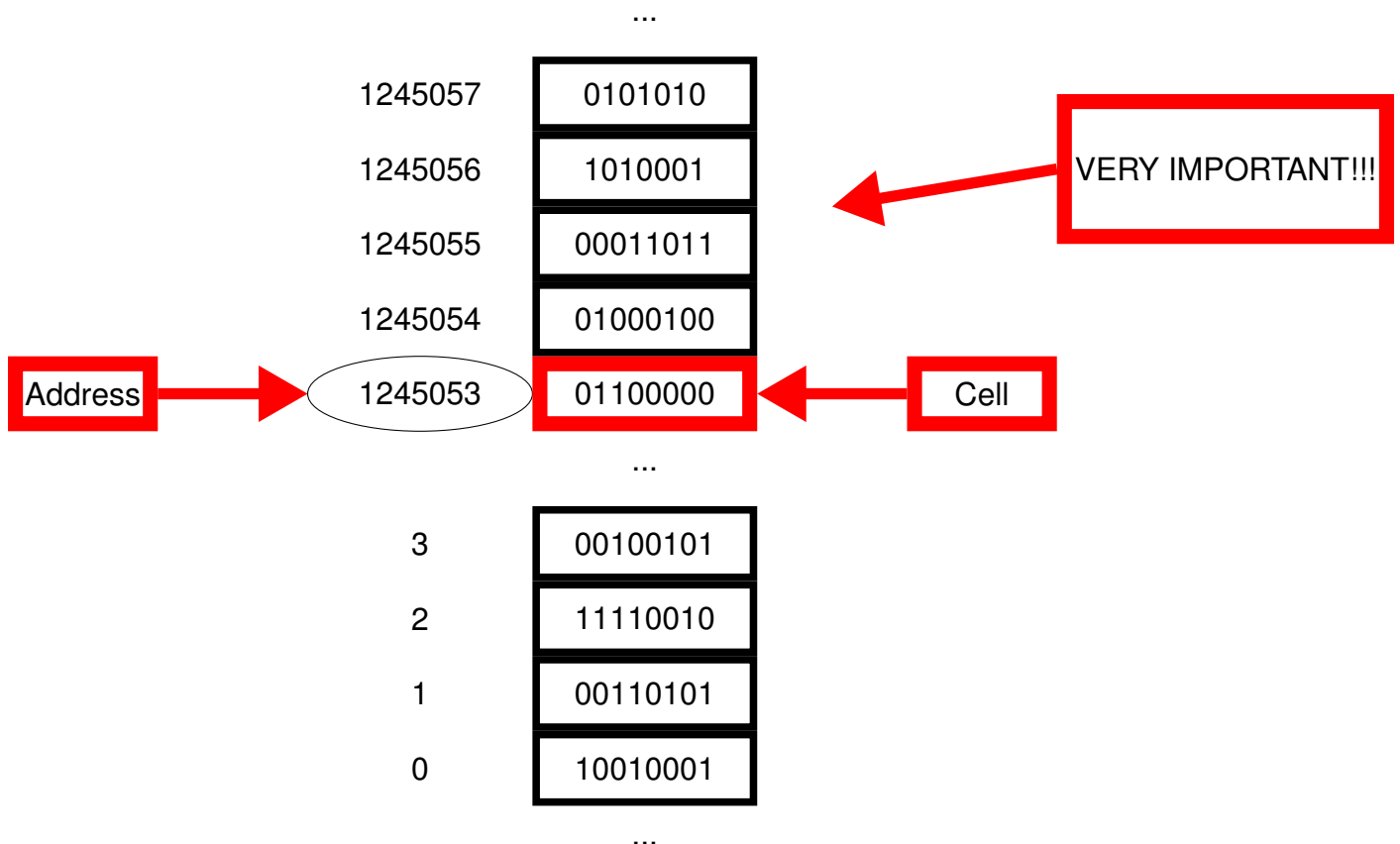
Here we go ... !!!

Memory

So far the model of the space where variables live does not have any organization. It's just a collection of cells with names and values.

Now for a more accurate model . . .

- A computer's memory is made up of a linear collection of memory cells - each is a byte (i.e. 8 bits)
- Each memory cell has a numeric address, its memory address
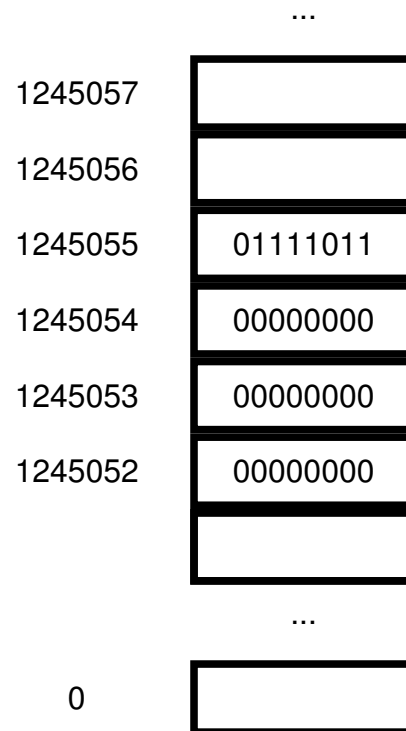- Each memory cell can hold 1 byte of data (1 byte = 8 bits)

```
                          ...

          1245057      0101010

          1245056      1010001              ◄──── VERY IMPORTANT!!!

          1245055      00011011

          1245054      01000100

  Address ──► (1245053)  01100000  ◄──── Cell

                          ...

                3        00100101

                2        11110010

                1        00110101

                0        10010001

                          ...
```

Remember: Each memory cell has **two** quantities, the **memory address** and the **content.**

Notice that memory addresses are **integer values**.

When you declare a variable, based on the type, C++ will allocate the correct number of bytes for that variable. For example, for an *int*, your

C++ (on a 32-bit machine) will allocate 4 bytes.

Also, when you declare two variables in the same scope, the memory used do not overlap – they occupy different memory spaces.

Memory addresses and memory address of a variable

In C++, if x is a variable, the **memory address** of x is **&x**.

Try this:

```
int x = 123;

std::cout << "val of x = " << x << '\n'
          << "addr of x = " << &x << '\n';
```

My output:

```
val of x is 123

addr of x is 1245052
```

(Your output is most probably going to be different from mine.) The **&** in the above is called the **address-of operator**.

Two things you must know right away . . .

First, technically speaking *&x* is the **address of the *value of x***. But I will sometimes call &x the **address of x** since that's the common practice.

Second, in reality, an `int` value takes up more than 1 byte. For a 32-bit machine, the value of x is spread out among 4 bytes: Recall that to see how many bytes are used to store an integer value you can do this:

```
std::cout << sizeof(int) << std::endl;
```

So in the above example when the program says that the address of x is 1245052, it means that the `int` value 123 occupies the memory at addresses 1245052, 1245053, 1245054, 1245055. The **address of x** actually refers to the **starting address**, i.e., address of the first byte, i.e. 1245052.

&x = address of first byte of the value of variable x

You don't have to worry about how the value 123 is "spread out" among 4 bytes. But briefly, the integer 123 (to human beings) is a bunch of bits (to a 32-bit computer):

$$00000000000000000000000001111011$$

(Details will be covered in CISS360.) So here's where you will find the value of our x from above:

...

1245057	
1245056	
1245055	01111011
1245054	00000000
1245053	00000000
1245052	00000000

...

| 0 | |

Notice that 123 is translated to 32 bits. Each byte can hold 8 bits. That's why the bits of an integer requires 4 bytes. At this point, we don't need to worry how the 123 is translated into bits. So I will usually simplify the above picture of your computer's memory by drawing this:

...

1245057	
1245056	
1245055	
1245054	
1245053	**123**
1245052	

...

0	
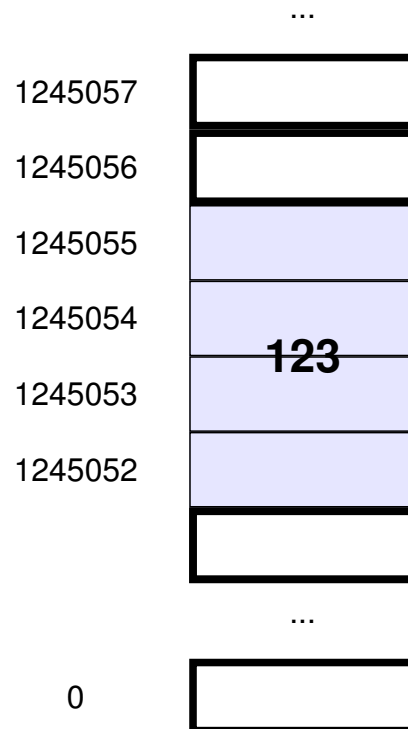
Let me summarize ...

- A computer's memory is made up of bytes.
- Associated to each byte is its address and its contents (value).
- The value of a variable occupies memory, possibly more than 1 byte.
- If `x` is a variable, then in C++
- `x` refers to the contents (value)
- `&x` refers to the beginning address of its value, i.e., the address of the first byte of its value

By the way the address printed when you run your program might contain things like a, c, e, etc. That's because the address is actually printed in **hexadecimals**, i.e., **numbers in base 16**. (You will also learn more about hexadecimals in CISS360.) Certain compilers allow you to typecast hexadecimal values into `int` value. Try the following (don't panic and call 911 if your compiler won't let you):

```
int x = 123;

std::cout << "val of x: " << x
          << '\n' << "addr of x: " << int(&x)
```

'An integer can be written (expressed) in many ways ... in
-base 10 or decimal (usual way)
-base 16 or hexadecimal
-base 8 or octals
-base 2 or bits'

```
            << '\n';
```

If it does not work, you can use any scientific calculator to convert from hex to decimals yourself. Most scientific calculators nowadays have the ability to convert between base 10 to base 2, 8, and 16.

In almost all programming languages, a hexadecimal number starts with 0x. So you might see something like this when working with address values:

<div align="center">0x013251a2</div>

The actual base 16 hexadecimal is 13251a2.

Technically speaking the memory address is an **unsigned integer**, i.e., it does not have a negative sign, i.e., it's a positive integer. So you can convert your memory address value to an unsigned integer like this:

```
int x = 123;
std::cout << "val of x: " << x
          << '\n' << "addr of x: "
          << (unsigned int)(&x)
          << '\n';
```

An **unsigned int** can only represent integers up to $2^{32} - 1$. If you have a newer laptop, your address values are very likely larger than $2^{32} - 1$. If your compiler yells at you, instead of type casting with **unsigned int**, use **unsigned long long int**.

An **unsigned int** is really a type. So you can declare an **unsigned int** variable. By the way, you have actually already seen **unsigned int** when using **srand()**. Likewise **unsigned long long int** is also a type.

Exercise -1.0.1. Find a scientific calculator program that can convert between base 10 and base 2, 8, and 16. (If you have time, use the web and learn to convert between base 10 and base 2, 8, 16 by hand – this will be covered in detail in CISS360.)

- Convert 42 into base 2, 6, 16.
- Convert base 16 a0234d1 to base 10.

Exercise -1.0.2. Declare two integer variables like this:

```
int x;
int y;
```

and print their addresses (in base 10). How far apart are the addresses of the two integer values of the variables? (Use your scientific calculator for subtracting two hexadecimal address values if you need to.) If the address differs by 4 (and the sizeof int is 4), then there's no gap between the memory occupied by the first and the second since an integer occupies 4 bytes. This means that your compiler is very efficient and is putting the value of `x` and `y` next to each other. (The address of the second variable is probably smaller.)

Exercise -1.0.3. This simple exercise is meant to make your brain remember `unsigned int`.

1. Declare an `unsigned int` variable `i` and initialize it with value 42. Print `i`. Get a positive integer from the user and give this value to `i`. Print `i`.
2. Write a simple program that computes the sum $1 + 2 + \ldots + i$ where `i` is provided by the user. Print the sum. In your program, since all integer values used are positive, you must only use `unsigned int` variables.

Exercise -1.0.4. Declare a `double` variable `x`. Print it's address. Now declare another `double` variable `y` and print the address of `y`. There shouldn't be a gap between the `double` values of the above variables if they are declared next to each other. From the program can you deduce the number of bytes used to store a `double`? Verify your guess by executing this:

```
double x = 3.14159;
std::cout << sizeof(double) << std::endl;
std::cout << sizeof(x) << std::endl;
```

Exercise -1.0.5. 1. How many bytes does a character variable/value use?
2. How many bytes does a boolean variable/value use?

Exercise -1.0.6. Declare an array of integers:

```
int x[3];
```

and print their address of `x[0]` and `x[1]` and `x[2]`. How far apart are they? Are the addresses ascending or descending?

WARNING!!!

Note that `&` is now used in two totally different ways!!!

```
int x;
int & y = x;     // int & is a type
std::cout << &x; // & is the address-of operator
```

Remember that!

Pointer Variable

At this point, you should ask the following question: "If you can put `int` values into `int` variables, `double` values into `double` variables, `bool` values into `bool` variables ... surely I can put address values into variables, right?" (If you did not ask that, then you're not learning actively! Wake up!)

So here we go ...
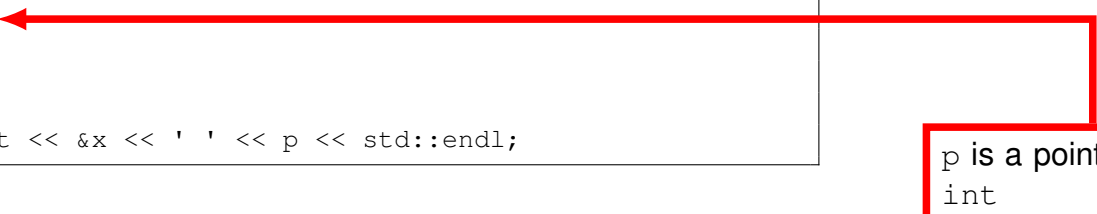
A **pointer variable** is a variable whose value is a memory address. Run this:

```
int x = 1;

int * p;

p = &x;

std::cout << &x << ' ' << p << std::endl;
```

`p` is a pointer to an `int`

We say that `p` **points to** `x` (well ... technically, `p` points to the **value** of `x`.) To declare a pointer variable you use this format:

```
[type] * [pointer];
```

You can of course do this, i.e. declaration with initialization:

```
int x = 1;
int * p = &x;
std::cout << &x << ' ' << p << std::endl;
```

Here's the format of a statement that declares a pointer with initialization:

```
[type] * [pointer] = [address of same type];
```

A very common convention is to begin the name of a pointer with `p` or `p_`. It's also common to include the name of the concept the pointer points to in the name:

Exercise -1.0.7. Complete this code segment. `psalary` should point to `salary`:

```
double salary = 100.23;

_____ * psalary = _____;
```

Verify!!! ... i.e., print the address of `salary` and `psalary` and make sure they are the same.

Exercise -1.0.8. Can you assign the address of a type of value to a pointer variable of another type? Answer: _____. Try this:

```
double x = 1.2;
int * p = &x; // trying to point an int pointer
              // to a double
```

You **cannot** initialize a pointer with the pointer value of a **different type**.

Exercise -1.0.9. Here's an `int` array:

```
int x[] = {2, 3, 5, 7, 11, 13};
```

Declare an `int` pointer, say call it `p`, that points to the value 11 in the array. Verify! Next, print the address of the value 13 and 7 in `x`.

Exercise -1.0.10. Here's a C-string:

```
char x[] = "hello world";
```

Declare a `char` pointer, say call it `p`, that points to the 'w' in the string. Verify!

Mental Picture

Look at this code:

```
int x = 42;
int * p;
p = &x;
```

Suppose the `int` value of x is at memory location 124500 in the computer and the memory location is at 124504. The computer's memory would look like this:
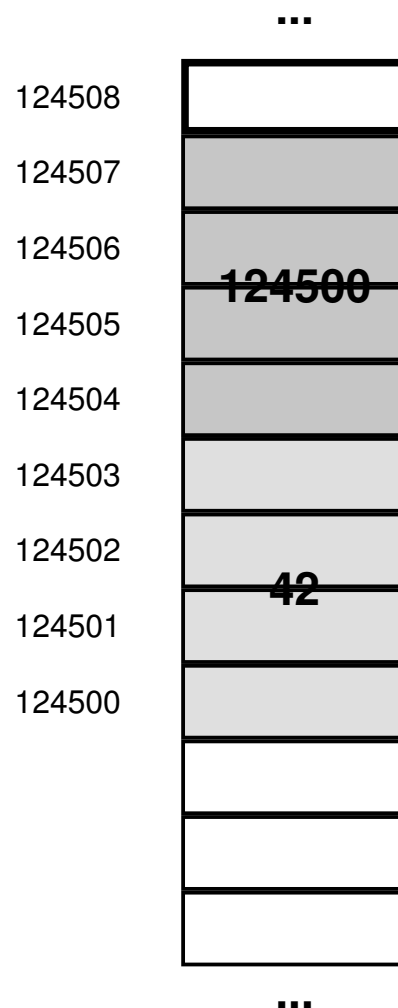
Here's the memory model that you are already used to:

x	42
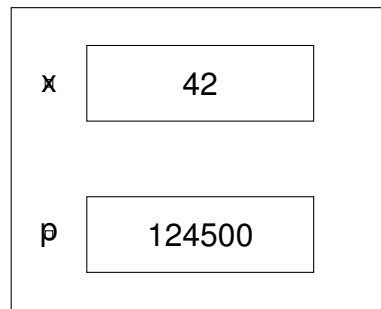
p	124500

To indicate that the value 42 of variable x is at memory address 124500, I will draw this:

124500

x	42

p	124500
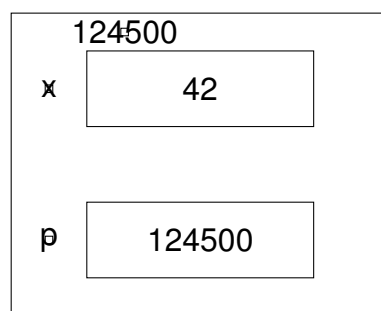
Well ... actually important thing is the **relationship** between p and the value of x: Think of p as knowing where to find the value of x. So we usually have this mental picture in mind when we talk about pointers:

124500

x	42

p	124500
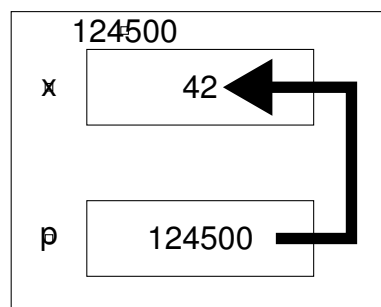
Sometimes, I will even simplify the mental picture by drawing this:
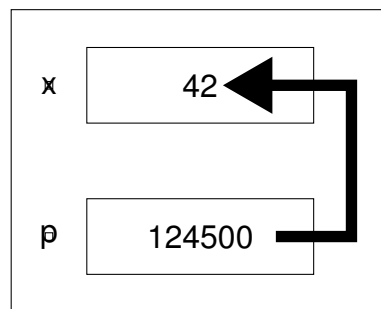
(Later, we'll see how to access access the value of x using p.)

Note that the arrow drawn in the diagram is to help us (human beings) see quickly what value a pointer is pointing to. Your computer only works with numbers (as in bits). The arrows in the drawing above are only a visual guide for us to see pointer relations quickly. There are no "hardware arrows" in your computer or CPU!!! Remember that!!!

Exercise -1.0.11. In a code fragment, there are two variables `i` and `j`, both of `int` type. The value of `i` is 5000 and the value of `j` is 6000. The value of `i` is at address 108080 and the address of the value of `j` is 501200. In addition, there are two pointer variables `p` and `q` such that `p` points to the value of `j` and `q` points to the value of `i`. Draw the memory model labeling everything (variable name, value, and address). Include arrows to relate pointers to the values the pointers point to.

Exercise -1.0.12. Same as above except that both `p` and `q` points to the value of `j`.

Exercise -1.0.13. Draw the memory model at the end of the following code fragment where the addresses of `pi[0]`, `pi[1]`, `pi[2]`, `pi[3]`, `pi[4]` are 8000, 8008, 8016, 8024, 8032, 8040.

```
double pi[] = {3.1, 3.14, 3.141, 3.1415, 3.14159};

double * p = &pi[1];
```

Exercise -1.0.14. Draw the memory model at the end of the following code fragment:

```
int x[] = {2, 3, 5, 7, 11};

int * p = &x[3];

int * q = &x[1];
```

(In this case, addresses of `x[0]`, `x[1]`, ... are not given, so just draw the simplified mental picture.)

Exercise -1.0.15. Given the following:

```
int x = 42; // address is 5000

double y = 3.1415; // address is 5004

char z = '$'; // address is 5012

bool b = true; // address is 5016
```

Declare 4 pointers (of the right type!) to point the each of the above variables. Draw the memory model.

Exercise -1.0.16. Write a code fragment so that you get the following memory model:
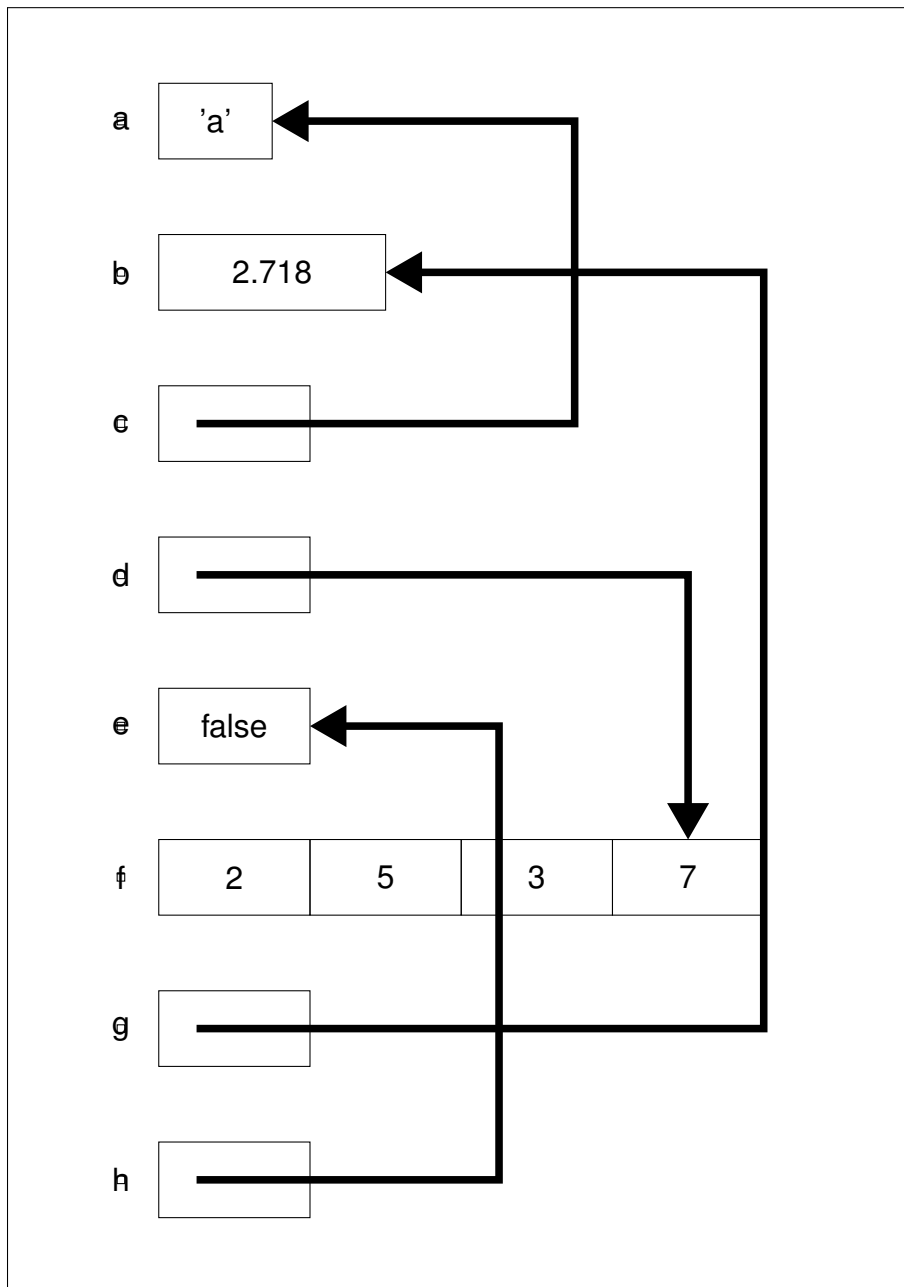
Common Gotcha

This is a very common gotcha:

```
int x = 0, y = 1;
int * p = &x, q = &y, r = &x;
```

It won't work (does not compile) because the above actually means this:

```
int x = 0, y = 1;
int * p = \&x;
int q = \&y;
int r = \&x;
```

It should be

```
int x = 0, y = 1;
int * p = \&x, * q = \&y, * r = \&x;
```

or (better) just write this:

```
int x = 0, y = 1;
int * p = \&x;
int * q = \&y;
int * r = \&x;
```

Memory address values and NULL

Up to this point I never assign constant address values directly to pointer variables like this:

```
int x = 42;         // say the value of x is at
                    // address 1205000
int * p = 1205000;  // p points to the value of x
```

To begin with, the address of x is usually different each time you run your program and is definitely different on a different computer. So the above would be a terrible idea.

In general, assigning a constant to a pointer variable is rare. This happens when you're doing something very special such as very low level programming where you need to access hardware or doing embedded systems programming. For instance, say you know that an `unsigned int` value at memory location 20500080 is used for communication with a hardware device, you would do something like this:

```
unsigned int * device = (unsigned int *) 20500080;
```

After that, whenever the device outputs a value, it's sent to the `unsigned int` at this location and `device` pointer will be used to access this value.

Unless otherwise stated, for CISS240, 245, 350, you will never assign or work directly with a constant memory address value. The only exception is the memory address **0**. However instead of using memory address 0 like this:

```
int * p = 0;
double * q = 0;
char * r = 0;
```

you use the predefined constant **NULL** like this:

```
int * p = NULL;
double * q = NULL;
char * r = NULL;
```

In order to use `NULL`, you might have to `#include <cstddef>` at the top of your code:

```
#include <iostream>

#include <cstddef>
```

```
int main()
{
    int * p = NULL;
    double * q = NULL;
    char * r = NULL;
    return 0;
}
```

Just to keep everything uniform, just make sure you include cstddef.

So what is the value at address 0???

Well the only important thing you need to know is this: **you are not allowed to access the value at address 0.** Therefore ...

NULL is only used to indicate that a pointer variable is not pointing to anything useful. (You can also use **nullptr** instead of **NULL**. For C++11 compliant compilers, **nullptr** and **NULL** are the same. C++11 ISO/IEC 14882 was released 9/2011.)

Because of the above, NULL is sometimes used to initialize a pointer variable before it is being used:

```
int x = 1;
int * p = NULL;   // p does not point to anything yet

... p is not used for anything useful ...

p = \&x;          // now point p to x

... now p is used to do something useful ...
```

Exercise -1.0.17. Print NULL. (Don't forget to type cast.) What do you see? Remember!

For a pointer p with NULL value, the picture is of course like this:

p | 0 |
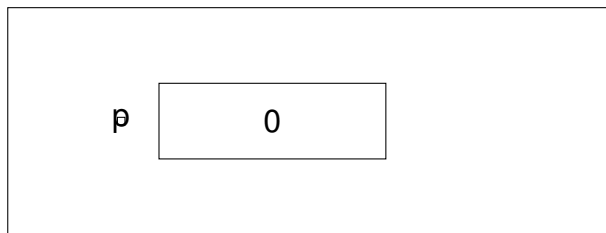
It's common to draw it like this:

p   [⊠]

or like this:

p   [____]———⏚
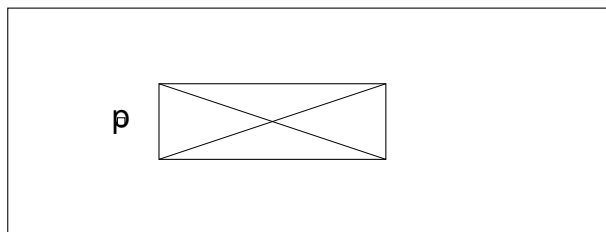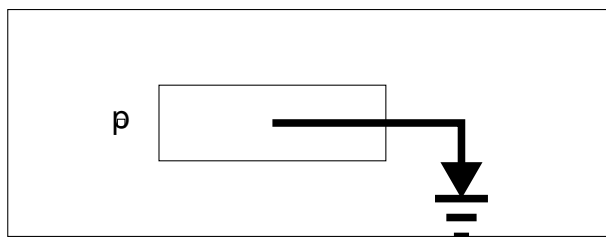
Dereferencing Pointers

Recall: If `p` is a pointer, then the value of `p` is the **memory address** of the value `p` is pointing to.

You can access the **value** `p` is pointing to. That's call **dereferencing** `p` and you do it using ***p**. The `*` used in this way is called the **dereferencing operator**. It's also called the indirection operator.
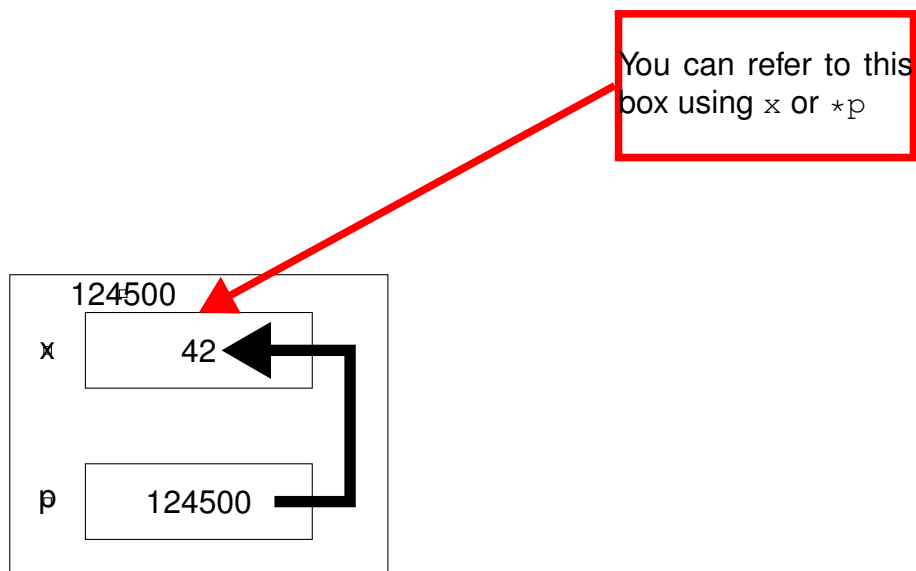
Try this:

```
int x = 42;
int * p;
p = \&x;

std::cout << x << ' ' << *p << '\n';

x = 0;
std::cout << x << ' ' << *p << '\n';

*p = 42;
std::cout << x << ' ' << *p << '\n';
```

Suppose `&x` is 124500, then the memory model looks like this:

MAKE SURE YOU SEE THE DIFFERENCE BE-TWEEN `p` and `*p`!!!:

- **`p` is 124500**
- **`*p` is 42**

WARNING:

There are now **3 different meanings for ***!!!

```
int x = 2 * 3;          // * = multiplication
int * p = &x;           // * = for pointer type
std::cout << x
         << ',' << *p   // * = for dereferencing
         << '\n';
```

It's really important to remember that if a pointer `p` points to a box, then the dereference of `p`, i.e. `*p`, refer to the contents or the "inside" of the box.

In particular, if I give you this picture without showing you the name of the variable with a value of 23:
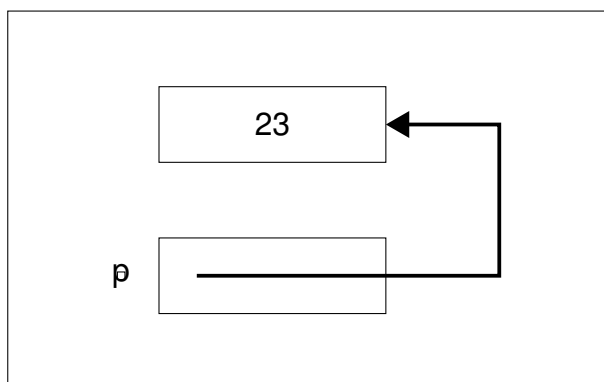
you tell me right away that `*p` is 23. You don't need to know the variable name of that box.

Note that `*p` refers to the contents of the box `p` points to in two ways and can be used in two ways, for reading and for writing:

- you can use `*p` to **read** the value that `p` points to
- you can use `*p` to **overwrite** the value `p` points to

For instance:

```
int x = 42;
int * p = &x;
int i;
i = *p;          // read the value that p points to (and
```

```
                // give it to i)
*p = 100;       // write the value (i.e. 100) onto the
                // value that p points to
```

So `*p` can appear on either the right or the left of the assignment operator.

Exercise -1.0.18. An **lvalue** is something that can appear on the left of = (the assignment operator). An **rvalue** is something that can appear on the the right of =. You have just seen that `*p` is is both an lvalue and an rvalue.

- Is 42 an lvalue? An rvalue?
- Suppose `x` is a variable. Is `x` an lvalue? An rvalue?
- Suppose `c` is a constant. Is `c` an lvalue? An rvalue?
- Suppose `f()` is a function with void return type. Is `f` an lvalue? An rvalue?
- Suppose `f()` is a function with `int` return type. Is `f` an lvalue? An rvalue?

Let me repeat the above. Suppose the address value in `p` is 124500. Then if you see something like

```
a = b + *p + c;
```

you should think of it as

```
a = b + (the value at address 124500) + c;
```

And if you see

```
*p = b + c;
```

you should think of it as

```
store value of (b + c) at address 124500
```

In summary,

1	2	3
2	2	5
5	2	
-1	2	6
1	2	3

Exercise -1.0.19. Add one statement to the following:

```
int x = 0;
int * p = &x;

// add one statement here to change the value of x
// to 42. Do NOT use the variable name x in your
// statement.

std::cout << x << '\n';
```

Exercise -1.0.20. Add one statement to this code fragment so that the output is 3.1415:

```
// p is a pointer variable that points to a double

// add one statement below this line



std::cout << *p << '\n';
```

Exercise -1.0.21. What is the output of this program:

```
int x = 2;
int * p = &x;
*p = *p + *p;
std::cout << x << '\n';
```

(Hint: Draw a picture)

Exercise -1.0.22. What is the output of this program?

```
int x = 2;
int y = 42;
int * p = &x;
int * q = &y;
std::cout << *p + *q << '\n';
```

(Hint: Draw a picture)

Exercise -1.0.23. What is the output of this program?

```
int x = 2;
```

```
int y = 42;
int * p = &x;
int * q = &y;
*p = *p * 2 + *q;
std::cout << x << ' ' << y << '\n';
```

(Hint: Draw a ...)

Exercise -1.0.24. What is the output of this program?

```
int x = 2;
int y = 42;
int * p = &x;
int * q = &y;
*p = *p * *q;
std::cout << x << ' ' << y << '\n';
```

(Hint: Draw, draw, draw ...)

Exercise -1.0.25. What is the output of this program?

```
int x = 2;
int y = 42;
int * p = &x;
int * q = &x;
*p = *p * *q;
*q = 42;
std::cout << x << ' ' << y << '\n';
std::cout << *p << ' ' << *q << '\n';
```

(Hint: Draw...)

Exercise -1.0.26. Do a simple addition program using the following skeleton.

```
int x = 0, y = 0;
int * p = &x;
int * q = &y;
// Statement to prompt the user for an integer and
// put it into x. Do NOT use x in your statement.

// Statement to prompt the user for an integer and
// put it into y. Do NOT use y in your statement.

std::cout << x + y << '\n';
```

Exercise -1.0.27. Declare a `double` variable called `d` and initialize it to 3.14. Declare a `double` pointer to point to `d`; call it `q`. Print `d` and `*q`. Add 1 to the value of `d` using `d`. Print `d` and `*q`. Add 1 to the value of `d` but using `q`. Print `d` and `q`.

Exercise -1.0.28. What is the output of this code fragment:

```
int x[] = {5, 4, 3, 2, 1};

for (int i = 4; i >= 0; -\/-i)
{
    int * p = &x[i];
    std::cout << (*p) << std::endl;
}
```

Exercise -1.0.29. Here's a program you are familiar with:

```
int s = 0;

for (int i = 0; i <= 100; i++)
{
    s += i;
    std::cout << i << ' ' << s << '\n';
}

std::cout << "1 + ... + 100 = " << s << '\n';
```

I'm going to add a pointer:

```
int s = 0;
int * p = &s;
for (int i = 0; i <= 100; i++)
{
    s += i;
    std::cout << i << ' ' << s << '\n';
}

std::cout << "1 + ... + 100 = " << s << '\n';
```

Now modify the program so that it works the same as before but the variable name `s` is not used after the statement that declares `p`:

```
int s = 0;
int * p = &s;

// Modify code below so that variable name s is not
// used.
```

```
for (int i = 0; i < 100; i++)

{
    s += i;
    std::cout << i << ' ' << s << '\n';
}

std::cout << "1 + ... + 100 = " << s << '\n';
```

Exercise -1.0.30. If you see this picture
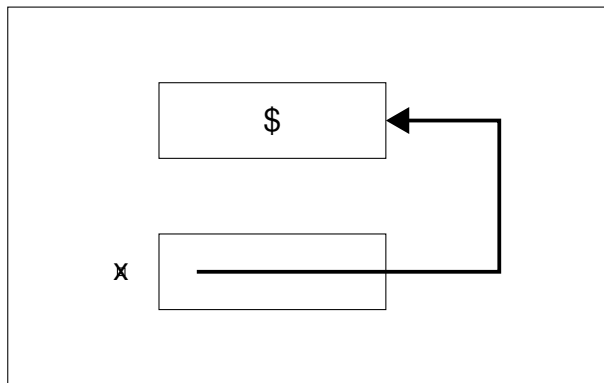
can you tell me what is the type of x?

Exercise -1.0.31. If you see this

```
double x = a[*b];
```

can you tell me what is the type of b?

Exercise -1.0.32. If you see this

```
std::cout << x % (*y);
```

can you tell me what is the type of y?

Exercise -1.0.33. Complete this:

```
void nextprime(int * p)
{

}

int main()
```

```
{
    int x = 7;
    nextprime(&x); // x becomes 11
    std::cout << x << std::endl;
    nextprime(&x); // x becomes 13
    std::cout << x << std::endl;
    nextprime(&x); // x becomes 17
    std::cout << x << std::endl;
    return 0;
}
```

Exercise -1.0.34. Can you apply the pre- and post-increment opera-tors to `*p` where `p` is a pointer to an `int`?

```
++(*p);

(*p)++;

--(*p);

(*p)--;
```

Do you need the parentheses? Can you apply the augmented as-signment operators to `*p`?

Note that if you do not assign a memory address to your pointer, it would have an arbitrary address value, pointing to some arbitrary value. Reading that value might result result in printing something that looks like garbage. It might even cause your program to crash if you do not have access rights to that location!!!

Another thing to note is that the dereferencing operator works with an address – it does not have to be applied to a pointer. In other words if you're doing this:

```
std::cout << *p << std::endl;
```

and if you know that `p` has value 124500, then the above can also be

```
std::cout << *124500 << std::endl;
```

Exercise -1.0.35. First get a strong cup of coffee ... now ... What is the output? Or is there an error?

```
#include <iostream>

int * f(int * x)
```

```
{
    return x;
}

int f(int x)
{
    return x;
}

int main()
{
    int a = 21;
    std::cout << *f(&a) + f(a) << std::endl;
    return 0;
}
```

Verify! (Draw a picture if you need to.)

Exercise -1.0.36. Sip your coffee ... another one ... What is the output? Or is there an error?

```
#include <iostream>

int h(int * z)
{
    return *z;
}

int * g(int * y)
{
    return y;
}

int * f(int * x)
{
    return (&x == NULL ? NULL : g(x));
}

int main()
{
    int a = 21;
    std::cout << *f(&a) + h(&a) << std::endl;
    return 0;
}
```

Verify!

Assignment Operator

It's not too surprising that you can do assignments on pointers . . .

First try to guess what this does:

```
int x = 42;
int y = 123;

int * p = &x;
int * q = &y;
std::cout << *p << "\n";

p = q;
std::cout << *p << "\n";
```

Now run it and verify.

Here are some pictures. Up to this point:

```
int x = 42;
int y = 123;

int * p = &x;
int * q = &y;

std::cout << *p << "\n";
```
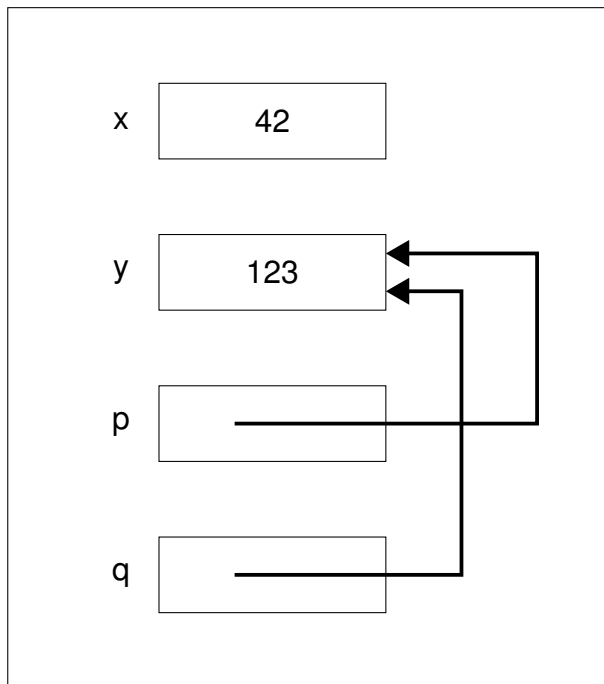
the memory model looks like this:
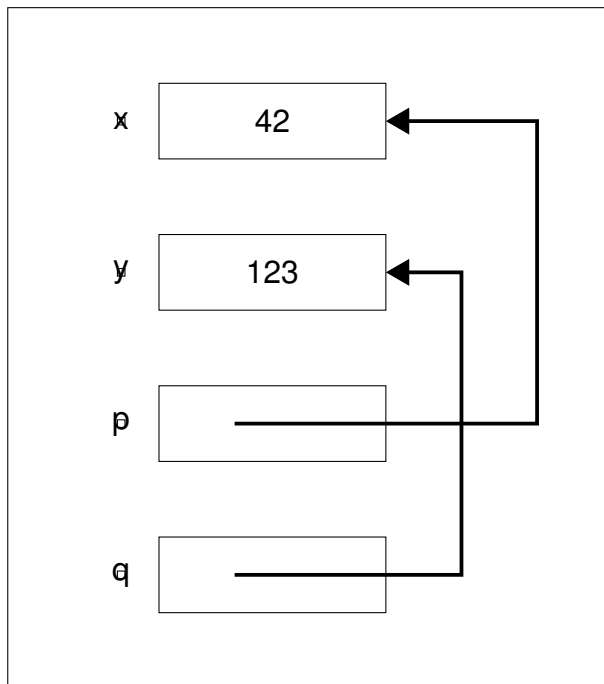
The next statement

```
p = q;
```

gives the memory address in q to p. Since q has the memory address of y, this means that p ends up with the memory address of y. So we get:

Since p points to y, or rather, p point to the `int` box that contains the value of y, of course *p gives us `123`.

Let me explain everything above all over again, this time using (made up) memory addresses ... **PAY ATTENTION!**

Here's the above code again, except that I've included some made-up addresses for the value of x and the value of y:

```
int x = 42;          // value of x is at address 80000
int y = 123;         // value of y is at address 80004

...
```

At this point in the code

```
int x = 42;          // value of x is at address 80000
int y = 123;         // value of y is at address 80004

int * p = &x;        // p = 80000
int * q = &y;        // q = 80004
std::cout << *p << "\n";
```

the picture looks like this:

The addresses are made up. Don't worry about them. I just gave them values of x and y addresses to illustrate the concepts below
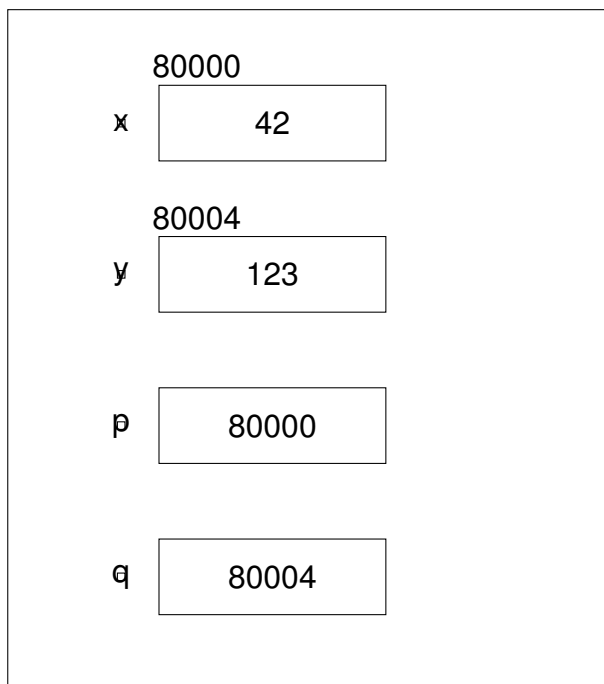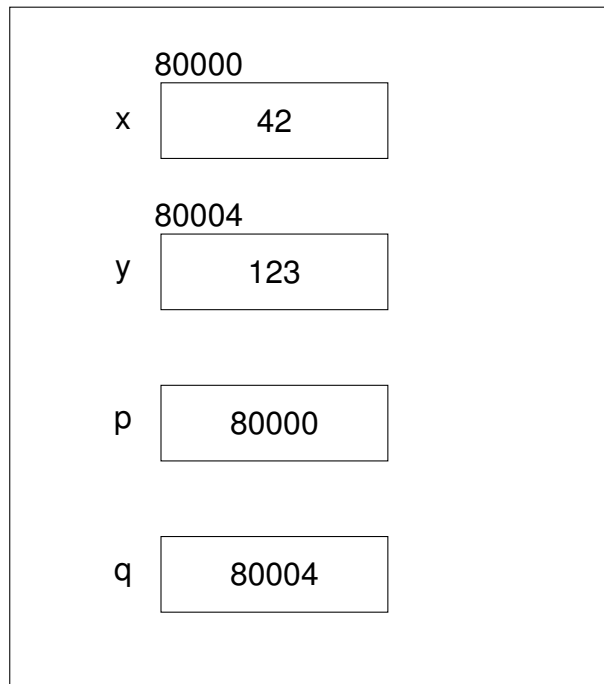
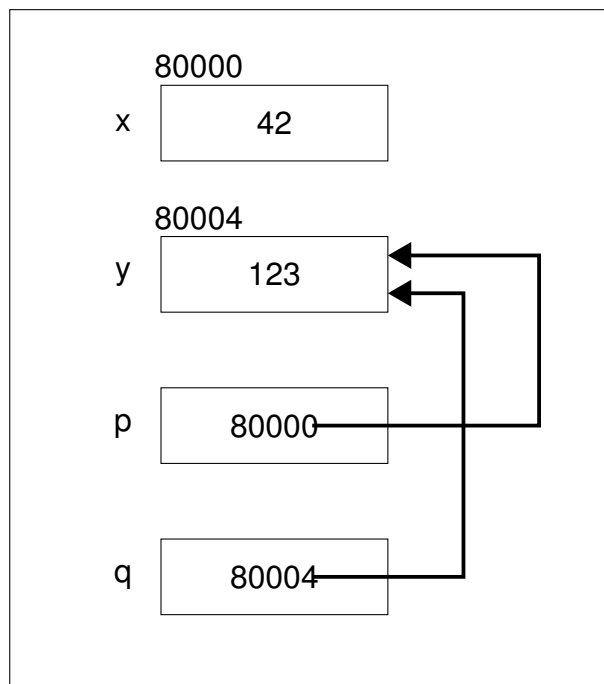and if I draw in arrows for pointers to help us see the pointer-to-value relationship it would look like this:
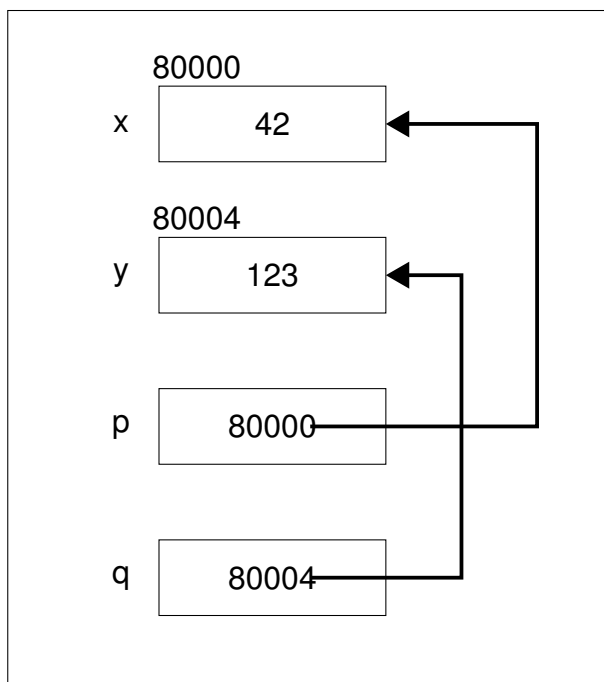
Next, the statement

```
p = q;
```

gives the value of q to p. The value of q is 80004. Therefore the value of p becomes 80004. The diagram now looks like this:

```
        80000
  x  ┌──────────┐
     │    42    │
     └──────────┘

        80004
  y  ┌──────────┐
     │   123    │
     └──────────┘


  p  ┌──────────┐
     │  80000   │
     └──────────┘


  q  ┌──────────┐
     │  80004   │
     └──────────┘
```

And if I now draw the arrows, I have

Get it?

Exercise -1.0.37. First draw memory models for this:

```
int x = 42;          // address 10000
int y = 43;          // address 10004
int * p = &x;
// Draw picture 1
std::cout << &x << ' ' << &y << ' ' << p << '\n';
p = &y;
// Draw picture 2
std::cout << &x << ' ' << &y << ' ' << p << '\n';
```

Verify your picture by running your program.

Exercise -1.0.38. Draw the memory models for the following:

```
int x = 42;          // address 80000
int y = 123;         // address 80004
int * p = &x;
int * q = &y;
// Draw picture 1

std::cout << &x << ' ' << &y << ' '
          << p << ' ' << q << '\n';
p = &y;
```

```
// Draw picture 2
std::cout << &x << ' ' << &y << ' '
          << p << ' ' << q << '\n';
q = &x;
// Draw picture 3
std::cout << &x << ' ' << &y << ' '
          << p << ' ' << q << '\n';
```

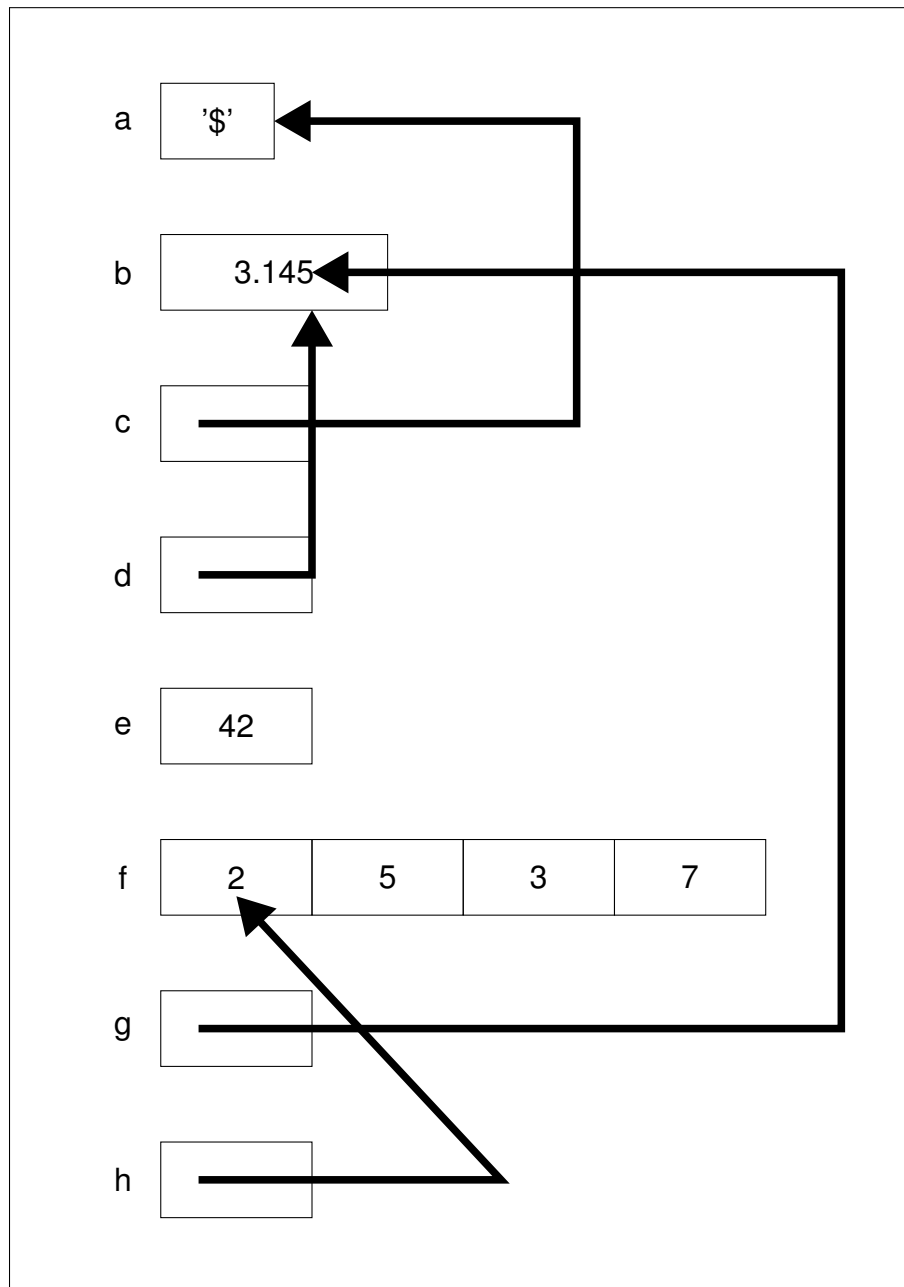**Exercise -1.0.39.** What is the output?

```
int i = 42;     // address 5000
int j = 5;      // address 5004
int k = -2;     // address 5008
int * p = &i;
int * q = &j;
int * r = p;
p = q;
q = r;
std::cout << *p << ' ' << *q << ' ' << *r << '\n';
```

(Hint: Draw a picture)

Exercise -1.0.40. You are brainstorming with your team in one of the company's meeting rooms. Your boss pops in to say hi on his way to get coffee and he notices the following diagram on the whiteboard. Someone is tracing a piece of code on the whiteboard:
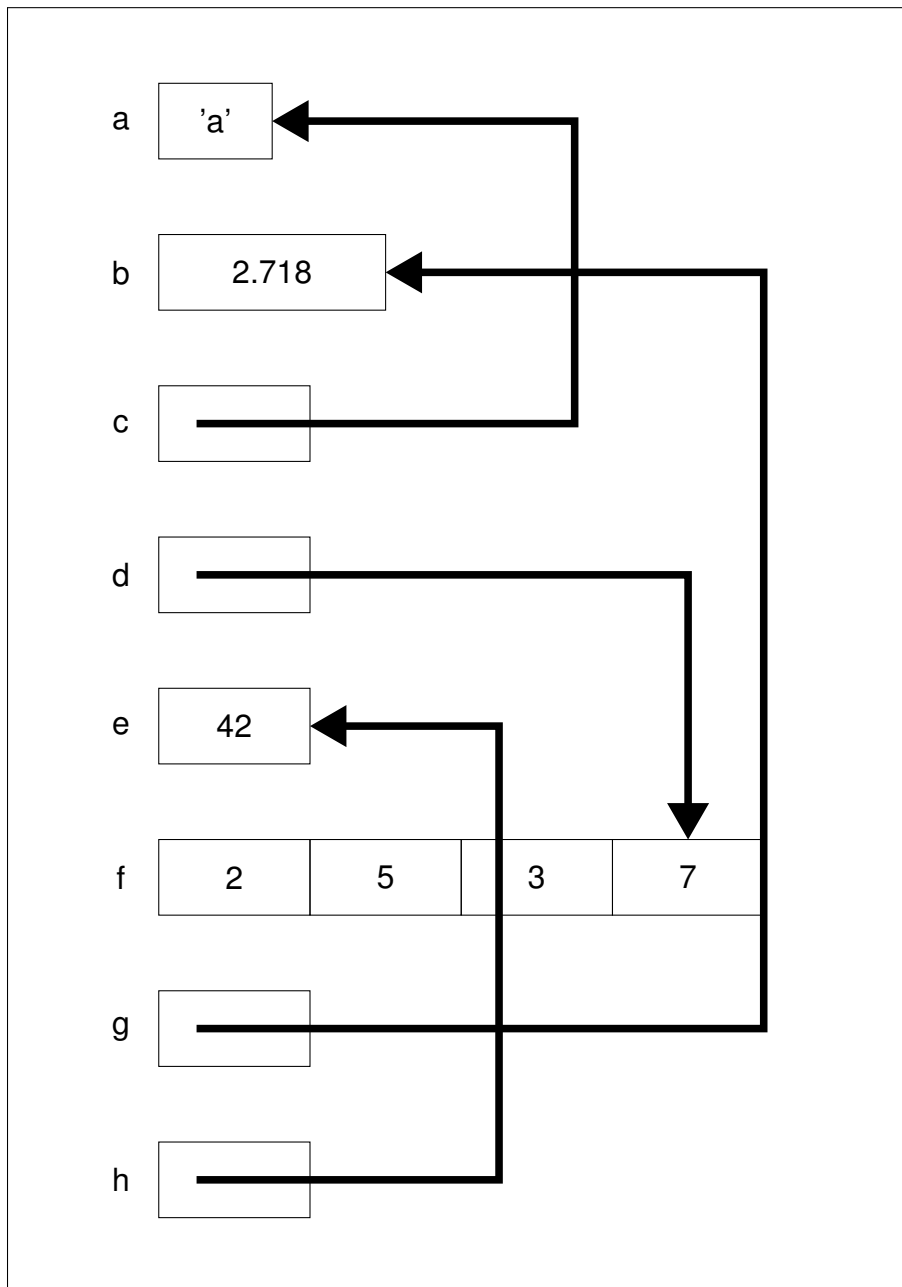
On his way back, your boss glanced at the whiteboard and sees this:

You notice he was shaking his head as he walked away. Why?

Exercise -1.0.41. What is the output?

```
int i = 1;
int j = 2;
int k = 3;
int * p = &k;
int * q = &i;
```

```
int * r = &j;
*p = *q + *r;
p = q;
*p = *q + *r;
p = r;
*p = *q + *r;
std::cout << i << ' ' << j << ' ' << k << '\n';
std::cout << *p << ' ' << *q << ' ' << *r << '\n';
```

Exercise -1.0.42. What is the output? Or is there an error?

```
int i = 1;
int j = 2;
int k = 3;
int * p = &k;
int * q = &i;
int * r = &j;
*p = *q + *r;
p = q;
*p = *q + *r;
p = r;
*p = *q + *r;
std::cout << i << ' ' << j << ' ' << k << '\n';
std::cout << *p << ' ' << *q << ' ' << *r << '\n';
```

Comparison

You can only compare pointers **of the same type.**

You can compare compatible pointers using

$$==, !=, <, <=, >, >=$$

because pointers are memory address and hence can be converted to integers and therefore can be compared just like integers. Try this

```
int x = 0;
int y = 1;
int * p = &x;
int * q = &y;
std::cout << (p == q) << "\n";
```

and this:

```
int x = 0;
char c = 'a';
int * p = &x;
char * r = &c;
std::cout << (p == r) << "\n";
```

Of course if two pointers point to the same memory address, then you get true when you compare their memory address values:

```
int x = 42;
int * p = &x;
int * q = &x;
std::cout << (p == q) << "\n";
```

Recall that variables declared in a block are lined up so that variables declared earlier will have a small address value.

```
int x = 0;
int y = 1;
int * p = &x;
int * q = &y;
std::cout << (unsigned int)(p) << ' '
          << (unsigned int)(q) << '\n';
std::cout << (p < q) << '\n';
```

In the above, you see that the address of x (i.e., the address of the value of x) is smaller that the address of y.

Exercise -1.0.43. TRUE or FALSE: The output is 1 since the value

that `p` and `q` points to are the same (i.e., 42).

```
int x = 42;
int y = 42;
int * p = &x;
int * q = &y;
std::cout << (p == q) << "\n";
```

Exercise -1.0.44. What is the output?

```
int x = 0;
int y = 1;
int z = 2;
int * p = &x;
int * q = &y;
int * r = &z;
*r = 0;
r = q;
*p = 0;
p = q;
*q = 0;
std::cout << (p == q) << ' ' << (q != r) << '\n';
```

Exercise -1.0.45. What is the output?

```
#include <iostream>

int f(int * x, int * y)
{
    return (x == y ? *x : *x + *y);
}

int main()
{
    int a = 2;
    int b = 3;
    std::cout << f(&a, &b) + f(&a, &a)
              << std::endl;
    return 0;
}
```

Exercise -1.0.46. What is the output? Or is there an error?

```
#include <iostream>

int f(int * x)
{
    return (x == NULL ? -1 : *x);
```

```
}

int main()
{
    int a = 2;
std::cout << f(&a) << ' ' << f(NULL)
        << std::endl;
    return 0;
}
```

[HINT: What is NULL again? What does it represent?]

Pointer Power #1: Assessing value anywhere

Here comes the first application of pointers ... a really important one
... we can use pointers to access a variable's memory anywhere we
like. (This is not new since references are exactly for that purpose.
You want to review references now.)

First of all recall the following:

```cpp
#include <iostream>

void swap(int x, int y)
{
    int t = x;
    x = y;
    y = t;
}

int main()
{
    int x = 0, y = 1;
    std::cout << x << ' ' << y << '\n';
    swap(x, y);
    std::cout << x << ' ' << y << '\n';
    return 0;
}
```

Of course the function uses pass-by-value. The `x` and `y` in `swap()`
has nothing to do with the `x` and `y` in `main()` and therefore `swap()`
cannot possibly change the values of variables in `main()`.

Now look at this:

```cpp
#include <iostream>

void swap(int & x, int & y)
{
    int t = x;
    x = y;
    y = t;
}

int main()
{
    int x = 0, y = 1;
    std::cout << x << ' ' << y << '\n';
    swap(x, y);
    std::cout << x << ' ' << y << '\n';
    return 0;
}
```

This second `swap()` function uses pass-by-reference and it does swap the values of two variables in `main()`.

With your understanding of pointers, run and explain this:

```cpp
#include <iostream>

void swap(int * x, int * y)
{
    int t = *x;
    *x = *y;
    *y = t;
}

int main()
{
    int x = 0, y = 1;
    std::cout << x << ' ' << y << '\n';
    swap(&x, &y);
    std::cout << x << ' ' << y << '\n';
    return 0;
}
```

Therefore pointers can be used as parameters to modify values of variables in the calling function, i.e., they can be used to achieve the same results as references. Actually references are secretly pointers!

Understand this very important idea: pointers allow you to access any value you like from anywhere in your code as long as know where to find the value (and you have access rights to that value):

```cpp
#include <iostream>

void j(int * x)
{
    *x = 42; // TADA!
}

void i(int * x)
{
    j(x);
}

void h(int * x)
{
    i(x);
```

```
}

void f(int * x)
{
    h(x);
}

int main ()
{
    int x = 0;
    f(&x);
    std::cout << x << std::endl;
    return 0;
}
```

Here's another example. This version of reciprocating x does not work:

```
#include <iostream>

void reciprocal(double x)
{
    x = 1.0 / x;
}

int main()
{
    double x = 2.0;
    reciprocal(x);
    std::cout << x << '\n';      // x is still 2.0
    return 0;
}
```

This one does work, using reference:

```
#include <iostream>

void reciprocal(double & x)
{
    x = 1.0 / x;
}

int main()
{
    double x = 2.0;
    reciprocal(x);
    std::cout << x << '\n';      // x is 0.5
    return 0;
}
```

And this one does work, using pointers:

```cpp
#include <iostream>

void reciprocal(double * x)
{
    *x = 1.0 / *x;
}

int main()
{
    double x = 2.0;
    reciprocal(&x);
    std::cout << x << '\n';      // x is 0.5
    return 0;
}
```

Now's your turn …

Exercise -1.0.47. Here's an increment function that does not work:

```cpp
#include <iostream>

void inc(int x)
{
    x++;
}

int main()
{
    int a = 42;
    inc(a);
    std::cout << a << std::endl;    // a is still 42
    return 0;
}
```

And here's one that works, using reference variables:

```cpp
#include <iostream>

void inc(int & x)
{
    x++;
}

int main()
{
    int a = 42;
    inc(a);
    std::cout << a << std::endl;     // a is 43
```

```
        return 0;
}
```

Complete the following program so that the new increment works, except that the parameter is a pointer and not a reference:

```
void inc(int * x)
{
}

int main()
{
    int a = 42;
    inc(&a);
    std::cout << a << std::endl;    // you should get 43
    return 0;
}
```

Exercise -1.0.48. Complete the `array_append()` function (which adds a value to the array `x` and increments the value of the length variable of the array. The expected output is `2 3 5 7`. d

```
#include <iostream>

void array_println(int x[], int xlen)
{
    for (int i = 0; i < xlen; ++i)
    {
        std::cout << x[i] << ' ';
    }
    std::cout << std::endl;
}

void array_append(int x[], int * xlen, int value)
{
}

int main()
{
    int x[10] = {2, 3, 5};
    int xlen = 3;
    array_append(x, &xlen, 7);
    array_println(x, xlen);
    return 0;
}
```

Exercise -1.0.49. Here's a function that acts like an integer input function:

```
#include <iostream>

void scanf(int * x)
{
}

int main()
{
    int a = 0;
    scanf(&a);
    std::cout << a << std::endl;
    return 0;
}
```

Complete this so that when you run it and enter 42, there's an output of 42.

Exercise -1.0.50. This is similar to the above:

```
#include <iostream>

void scanf(double * x)
{
}

int main()
{
    double a = 0;
    scanf(&a);
    std::cout << a << std::endl;
    return 0;
}
```

Test it!!!

Constantness for pointers

Recall that for a regular variable like this:

```
int x = 42;
```

to make `x` constant so that its value cannot change, you do this:

```
const int x = 42;
```

Making a variable constant is a protection mechanism to prevent change in the case when a variable should not be changed. (Make sure you review the notes on constants.)

The above is old stuff. Now for something new ...

Because a pointer is associated with two values, an address and a value the pointer points to, you can "const" two values associated with a pointer. First let me describe the ideas, then I'll talk about the syntax.

Look at the following picture of a pointer `p` pointing to 42 (at address 5002048):
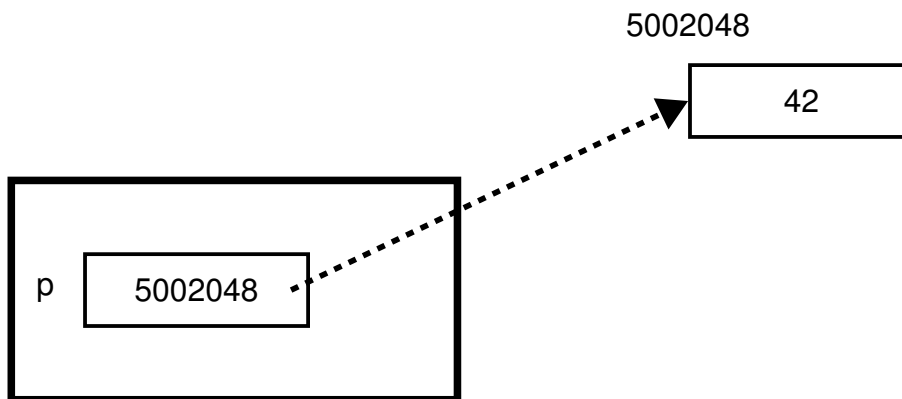
If `p` is a **constant pointer**, then the pointer value of `p` is a constant and therefore cannot change.

5002048

42

p | 5002048

Cannot be changed!!!!
p = ... is wrong!!!!
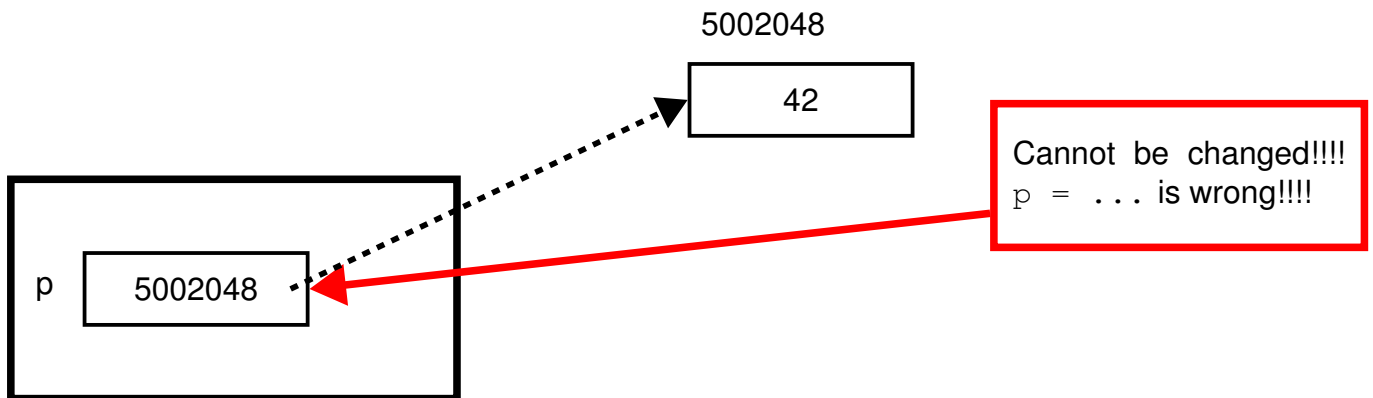
However if p is a **pointer to constant**, then the value that p points to cannot be changed through using *p:

5002048

42

p | 5002048

Cannot be changed!!!!
p = ... is wrong!!!!
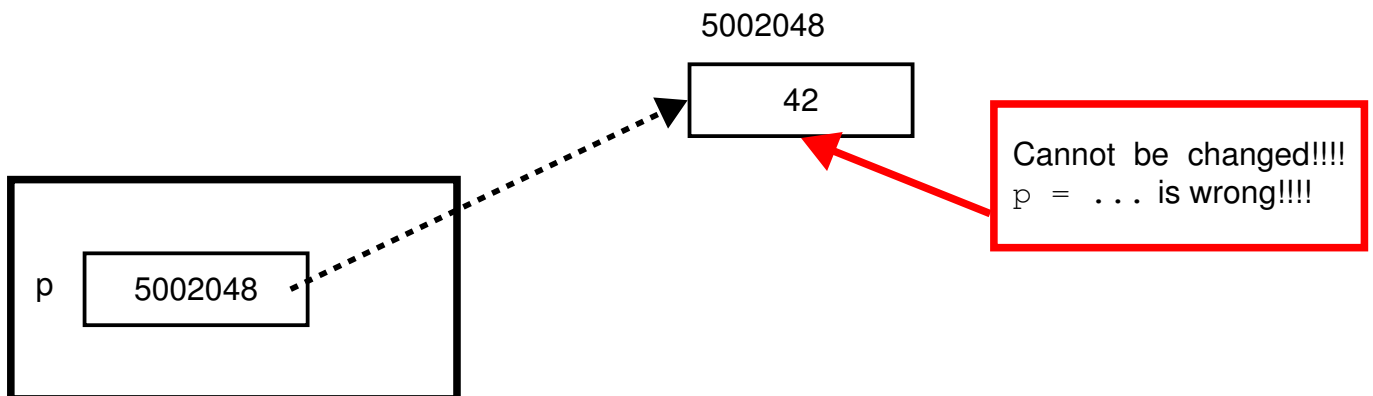
Easy right?

Now let's talk about the syntax . . .

In the case of a pointer, say:

```
int x = 42;
int * p = &x;
```

If you do not want the memory address value in p to be changed, you want p to be a **constant pointer**, you do this:

```
int x = 42;

int * const p = &x;
```

Note where const is placed: to the right of *.

Exercise -1.0.51. To make sure you remember the above, first run this

```
int x = 0, y = 1;
int * p = &x;
std::cout << *p << '\n';
p = &y;
std::cout << *p << '\n';
```

Everything works just fine. Now make `p` a constant pointer like this:

```
int x = 0, y = 1;
int * const p = &x;
std::cout << *p << '\n';
p = &y; // compiler complaints!!!
std::cout << *p << '\n';
```

Get it?

What about the case where `p` points to a constant value? First run this

```
int x = 0;
const int * p = &x;
std::cout << *p << '\n';
```

No problems. Now try this:

```
int x = 0;
const int * p = &x;
std::cout << *p << '\n';
*p = 42; // compiler complaints!!!
```

That's it!

There is however one very important thing you must remember. Run this:

```
int x = 0, y = 1;
const int * p = &x;
std::cout << x << ' ' << *p << '\n';
x = 42;                 // OK ... x is a variable
std::cout << x << ' ' << *p << '\n';
```

Note: **If `p` points to a `const int`, it doesn't mean that the `int` value can't change – it's just that you cannot change it using `*p`.** In the above, we change the value using `x` Get it?

Here is a table the summarizes all the cases:

		Can change p? Is p = ... OK?	Can change *p? Is *p = ... OK?
`int *`	`p`	YES	YES
`const int *`	`p`	YES	NO
`int * const`	`p`	NO	YES
`const int * const`	`p`	NO	NO

Here's an important **memory aid** (this might not be in your text-book): You **read** the pointer type **backwards**. For instance, if you see this declaration:

```
const int * const p
```

you read it in this direction:

```
const int * const p
```


so that you say "p is a constant pointer to an integer constant". Here are all the 4 possible constantness cases for a pointer declaration:

You say:

`int *`	`p`	"p is a pointer to int"
`const int *`	`p`	"p is a pointer to int const"
`int * const`	`p`	"p is a const pointer to int"
`const int * const`	`p`	"p is a const pointer to int const"

Two important things to note ...

A constant pointer must be initialized. (This is just like a regular constant integer variable – all constants must be initialized.)

This means:

```
int * const p;          // WRONG!!!
p = &x;                 // TOO LATE!!!

const int * const q;    // WRONG AGAIN!!
q = &y;                 // TOO LATE BOZO!!!
```

A non-constant pointer cannot point to a con-

stant. For instance

```
const int x = 42;
int * p0 = &x;       // WRONG!!!
int * const p1 = &x; // WRONG!!!
```

Why? Because in this case, `p` is supposed to point to an `int` (not `const int`) Therefore `p` can change the value that it points to using `*p`. But that value is a `const`! That's a contradiction!

Exercise -1.0.52. Is this OK?

```
const int x = 42;
const int * p = &x;
int * const q = p;
```

Here are some exercises to tease your brain.

Exercise -1.0.53. Does the following code fragment compile?

```
int x = 42, y = 24;
int * p = &x;
p = &y;    // OK?
*p = 42;   // OK?
```

Verify on your own.

Exercise -1.0.54. Does the following code fragment compile?

```
double x = 3.14, y = 2.71;
double * const p = &x;
p = &y; // OK?
*p = 0; // OK?
```

Verify on your own.

Exercise -1.0.55. Does the following code fragment compile?

```
char x = 'a', y = 'b';
const char * p = &x;
p = &y;     // OK?
*p = 0;     // OK?
```

Verify on your own.

Exercise -1.0.56. Does the following code fragment compile?

```
bool x = true, y = false;
const bool * const p = &x;

p = &y;      // OK?
*p = 0;      // OK?
```

Verify on your own.

Exercise -1.0.57. Which of the following pointer declarations with initialization works?

```
const char x = '#';
      char * p0 = &x;
      char * const p1 = &x;
const char * p2 = &x;
const char * const p3 = &x;
```

Exercise -1.0.58. Does this compile? If it does, what's the output?

```
double a = 3.14;
const double * p = &a;
*p = a + *p;

std::cout << a << ' ' << *p << '\n';
```

Exercise -1.0.59. Does this compile? If it does, what's the output?

```
double a = 3.14;
const double * p = &a;
a = a + *p;

std::cout << a << ' ' << *p << '\n';
```

Connection between pointers and references

The previous section shows you that the behavior of pass-by-references can be achieved by pass-by-value where the value passed into a function is a pointer value (i.e., address value).

As I've mentioned before when you use pass-by-reference, your compiler actually converts your function to one using pointers.

To illustrate the connection further run this program:

```
int x = 42;
int & r = x;    // r is a reference to x
int * p = &x;   // p is a pointer to x

std::cout << x << ' ' << r << ' ' << *p << '\n';

std::cout << &x << ' ' << &r << ' ' << p << '\n';
```

You will see that if `r` is a reference to `x`, the address of `r`, i.e. `&r`, is the same as `&x`. This means that the value `r` refers to resides at the address of `x`, i.e., the output of the second line contains the same address.

Let me repeat: The address of a reference variable is the address of the value the reference is referring to. Therefore if `r` reference x and `p` points to `x`, then

`&r` and `p` are the same

and

`r` and `*p` are the same.

Think, understand, and remember!

Differences between pointers and references

The above might lead you to think that pointers and references are the same (except for syntax). There are actually lots of differences between pointers and references. With your understanding of pointers, let's take a closer look . . .

1. There's a pointer value, NULL, that can be assigned to all pointers. This frequently acts as a default pointer value. Recall that NULL does not point to anything useful. References (of any type) MUST refer to a value of a variable (or constant) of the same type – there's no such thing as NULL reference. So a function that accepts a pointer has the option of being called with "nothing":

```cpp
void print_cash(int * p_cash_in_cents = NULL)
{
    if (p_cash_in_cents == NULL)
    {
        std::cout << "no cash!!!\n"
    }
    else
    {
        int dollar = *p_cash_in_cents / 100.0;
        int cents = *p_cash_in_cents - dollar * 100.0;
        std::cout << '$' << dollar
                  << '.' << cents << '\n';
    }
}
```

2. A pointer can be a constant pointer or non-constant pointer:

For a non-constant pointer, the address value in the pointer can change so that it can point to another value.

```cpp
p = &x; *p++; // increments x
p = &y; *p++; // increments y
p = &z; *p++; // increments z
```

```cpp
int x = 0, y = 0, z = 0;
int & r = x; r++;          // increments x
r = y; r++;                // increments y ... NO!!!
```

The fact that a pointer can point to different values is actually extremely important later when it comes to pointers and arrays.

Pointer Power #2: Speeding up function calls

Now for our second application of pointers (and references): speeding up function calls by passing in pointer values (or references) in order to avoid copying of values

First look at this very simple (and old) program:

```cpp
#include <iostream>

int max(int x, int y)
{
    return (x > y ? x : y);
}

int main()
{
    int a;
    std::cin >> a;
    int b;
    std::cin >> b;
    int x = max(a, b);
    std::cout << x << '\n';
    return 0;
}
```

No big deal.

Now run this

```cpp
#include <iostream>

int * max(int * x, int * y)
{
    if (*x > *y)
    {
        return x;
    }
    else
    {
        return y;
    }
}

int main()
{
    int a;
    std::cin >> a;
    int b;
    std::cin >> b;
```

```
    int * p = max(&a, &b);
    std::cout << *p << '\n';
    std::cout << &a << ' ' << &b
            << ' ' << p << '\n';
    return 0;
}
```

It's helpful if you draw a picture of the computation and explain what's happening. Don't panic ... just draw the picture and explain slowly and carefully.

You can clean up the `max()` function above like this:

```
int * max(int * x, int * y)
{
    return (*x > *y ? x : y);
}
```

This is what's going on in the program: instead of computing and returning the maximum of two values, the function actually returns a pointer value that points to the address of the variable with the larger value.

We can even do this: Note that the above max function has pointer parameters whose pointer values do not change – we can make them constant pointers. They also do not change the value they point to – we can make them point to constant integers. So we can do this:

```
#include <iostream>

const int * const max(const int * const x,
                      const int * const y)
{
    return (*x > *y ? x : y);
}

int main()
{
    int a;
    std::cin >> a;
    int b;
    std::cin >> b;
    const int * const p = max(&a, &b);
    std::cout << *p << '\n';
    std::cout << &a << ' ' << &b << ' '
            << p << '\n';
    return 0;
}
```

or this:

```
#include <iostream>

const int * const max(const int * const x,
                      const int * const y)
{
    return (*x > *y ? x : y);
}

int main()
{
    int a;
    std::cin >> a;
    int b;
    std::cin >> b;
    std::cout << *max(&a, &b) << '\n';
    std::cout << &a << ' ' << &b << ' '
                 << max(&a, &b) << '\n';
    return 0;
}
```

Note that you can also do this using references. Study the following program carefully:

```
#include <iostream>

const int & max(const int & x, const int & y)
{
    return (x > y ? x : y);
}

int main()
{
    int a;
    std::cin >> a;
    int b;
    std::cin >> b;
    const int & r = max(a, b);
    std::cout << r << '\n';
    std::cout << &a << ' ' << &b
                 << ' ' << &r << '\n';
    return 0;
}
```

Now why would you want to pass in and return a pointer (or references) to one of the variables that contains the maximum value and not compute and return the maximum value??? Isn't the first method a lot simpler?!?

Because the pointer (or reference method) can save memory (and potentially time). How so?

When you compute the maximum value and return that value, you have to create that value and put it somewhere for returning. (Recall the stack?) For returning the maximum of two doubles, I have to save a double value onto the stack – that's 8 bytes. For the pointer method (and similarly for the reference method), I have to save the pointer address of the variable holding the maximum value onto the stack. A pointer address is 32-bits (on a 32-bit machine) which is 4 bytes. Also, remember that using pass-by-value, the two doubles would have to saved onto the stack for the max function to retrieve.

And what if the two values I'm computing with actually takes more bytes? Then the difference is even greater. For instance in C++, there's a `long double` type. On my computer, the `long double` occupies 12 bytes. So for the method that returns the maximum value, I have to

- copy two `long double`s values (24 bytes) onto the stack
- compute and return a `long double` (12 bytes)

For the pointer method that returns an address, I have to

- copy two addresses (8 bytes) of `long double` variables onto the stack
- compute the address (4 bytes) for returning.

There's of course a cost involved for the pointer method – I have to dereference the pointers. But in any case, when the values to be used for computing occupies a huge amount of space, the pointer (or reference) method is preferred.

Exercise -1.0.60. This is a very important exercise!!! (A very common mistake):

```
int * max(int * x, int * y)
{
    int q;
    q = (*x > *y ? *x : *y);
    return &q;
}
```

What's the problem? (Draw a picture!!!) There's an analogous one

using references:

```
int & max(int x, int y)
{
    int q;
    q = (x > y ? x : y);
    return q;
}
```

Here's a general principle: **Never return a pointer that points to a local variable in a function!!!** Because the pointer will point to a value that will be destroyed when you exit the function!!! In the same way: **Never return a reference that refers to a local variable in a function!!!**

Protecting function from accidentally changing data

Before we go on to the next use of pointers, let's look at the previous section on functions that accept pointers and/or references.

Recall that if you have a function that accepts variables which are "big",

```
int f(long long int x, long double y)
{
    return (x > y ? 0 : 1);
}

int main()
{
    long long int x = 42;
    long double y = 3.14;
    int x = f(x, y);

    return 0;
}
```

Then the act of making a function call is faster if you pass in pointers (or references) because the data to be passed in is not copied, only the addresses of the data are passed in:

```
int f(long long int & px, long double & py)
{
    return (*px > *py ? 0 : 1);

}

int main()
{
    long long int x = 42;
    long double y = 3.14;
    f(&x, &y);

    return 0;
}
```

Note that the larger the data, the more crucial it is to use pointers or references.

If `f()` is meant to change the x and y back in `main()`, then of course you must pass in pointers or references.

However if `f()` is not meant to change `x` and `y` back in `main()`, then passing pointers (or references) into `f()` can be dangerous since f() can potentially change the `x` and `y` in `main()`. To overcome this, you pass in pointer-to-constants (or constant reference):

```
int f(const long long int * px,
      const long double * py)
{
    return (*px > *py ? 0 : 1);
}
```

or

```
void f(const long long int & x,
       const long double & y)
{
    return (x > y ? 0 : 1);
}
```

In general do the following for a function call `f(..., x, ...)` (say from `main()`) where `x` has type `T`:

```
void f(..., T x, ...)
{
    ... x ...
}

int main()
{
    T x;
    f(..., x, ...);
}
```

Do the following:

CASE 1. If `f()` needs to change the `x` in `main()`, then parameter `x` needs to be a reference **(pass-by-reference)**:

```
void f(..., T & x, ...)
{
    ... x ...
}

int main()
{
    T x;
    f(..., x, ...);
}
```

or parameter `x` should be changed to a pointer

```
void f(..., T * x, ...)
{
    ... *x ...
}

int main()
{
    T x;
    f(..., &x, ...);
}
```

CASE 2A. If `f()` does not need to change the x in `main()`, and x in `main()` does not use much memory, NO CHANGE IN ABOVE.

CASE 2B. If `f()` does not need to change the x in `main()`, and x in `main()` uses a lot of memory, then parameter x need to be a pointer to a constant:

```
void f(..., const T * const x, ...)
{
    ... *x ...
}

int main()
{
    T x;
    f(..., &x, ...);
}
```

Or you can use a constant reference: **(pass-by-constant-reference)**

```
void f(..., const T & x, ...)
{
    ... x ...
}

int main()
{
    T x;
    f(..., x, ...);
}
```

Chaining

Look at this (this is old stuff ... no biggy):

```
void inc(int * p)
{
    *p = *p + 1; // OR: ++(*p)
}

int main()
{
    int x = 0;
    inc(&x); inc(&x); inc(&x);
    std::cout << x << std::endl;
    return 0;
}
```

Now look at this version:

```
int * inc(int * p)
{
    *p = *p + 1;
    return p;
}

int main()
{
    int x = 0;
    inc(inc(inc(&x)));
    std::cout << x << std::endl;
    return 0;
}
```

Study the above code very carefully!!! (Chaining will be used a lot later when woirking with classes and objects.)

When a function accepts a pointer value and returns the same pointer value, that function allows chaining, i.e., calling the function multiple times using the same pointer value.

Study this version too (this one does not use references – the idea is the same):

```
int * const inc(int * const p)
{
    *p = *p + 1;
    return p;
}
```

```
int main()
{
    int x = 0;
    inc(inc(inc(&x)));
    std::cout << x << std::endl;
    return 0;
}
```

(i.e., the pointer can be constant.) Study this one that uses references::

```
int & inc(int & r)
{
    r = r + 1;
    return r;
}

int main()
{
    int x = 0;
    inc(inc(inc(x)));
    std::cout << x << std::endl;
    return 0;
}
```

Exercise -1.0.61. Now's your turn. Write functions that will allow this to happen:

```
int prime = 5;

nextprime(nextprime(nextprime(&prime))) // prime = 13
```

where `nextprime(p)` will set `*p` to the prime that is $> *p$. When you're done, do a version that uses pass-by-reference:

```
int prime = 5;

nextprime(nextprime(nextprime(prime))) // prime = 13
```

Exercise -1.0.62. How about this one:

```
int index = 0;

int x[] = {3, 5, 7, 9, 2};

inc(&index, x);
```

where `inc(&index, x)` increments `x[index]` and increment `index` by 1, and returns `&index`.

To increments the first three values of the array x, we can do this:

```
int index = 0;

int x[] = {3, 5, 7, 9, 2);

inc(inc(inc(&index, x), x), x);
```

Pointer Power #3: Dynamic memory management

Now we are ready to talk about **dynamic memory management.** For now I will focus on memory management for a single value of basic type. In the next set of notes, we'll talk about dynamic memory management for arrays.

In our examples like

```
int x = 1;

int * p = &x;
```

we always declare another variable for `p` to point to. You can also point `p` to an integer value which is **not** associated to an integer variable.

Try this:

```
int * p;

p = new int;

*p = 42;

std::cout << *p << std::endl;
```

or

```
int * p = new int;

*p = 42

std::cout << *p << std::endl;
```

You should think of

```
p = new int;
```

as creating an `int` value **somewhere** in the computer's memory and giving the memory address of this value to `p`. We say that we are **allocating memory** for `p` (to point to). The integer value created is **not** associated with an integer variable – there's no integer variable name involved:
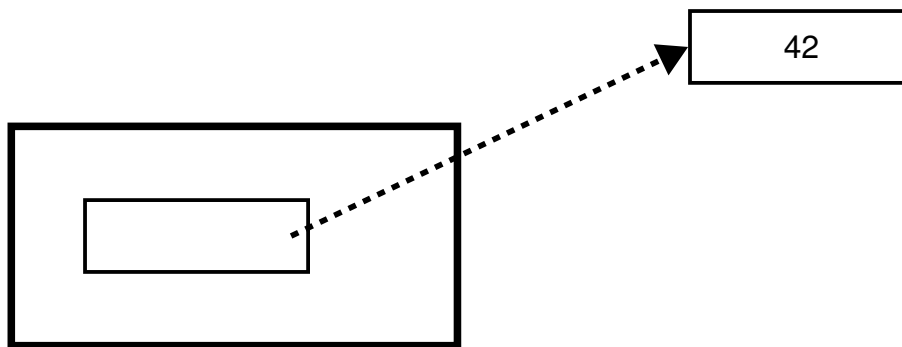
Look at the picture: It's REALLY important to understand that

- pointer `p` consumes memory and
- the value `p` points to also consumes memory.

Make sure you see the difference between the two. Not distinguishing the two is the cause of many confusion later on when you learn more about memory management.

Exercise -1.0.63. What is the output of this program if the user entered 5?

```
int * x = new int;
int * y = new int;
*y = 1;

std::cin >> *x;
*x = *x + *y;

std::cout << *x << ' ' << *y << '\n';
```

In general you can `new` any type: you can `new` a `double`, a `char`, a `bool`, etc.

Remember `NULL`? If you have declared a pointer `p`, but you have not allocated memory for `p`, then `*p` of course should not be used. And one way to remind yourself that `p` has not been allocated is to initialize it to `NULL`:

```
int * p = NULL;
                // etc.
                // now you need an integer \ldots

if (p == NULL)
{
    p = new int;
```

```
}
// Now use *p
```

Exercise -1.0.64. What is the output of this program?

```
int * x = new int;
*x = 1;
double * y = new double;

*y = 5.1;

for (int i = 0; i <= *y; i++)
{
    *x *= 2;
}
std::cout << *x << '\n';
```
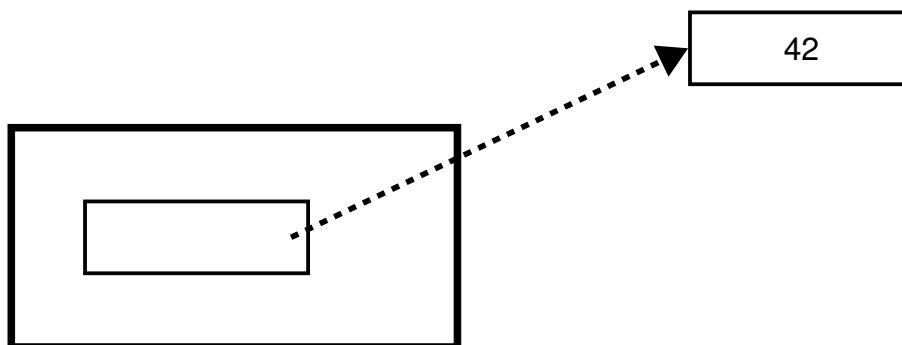
Exercise -1.0.65. What's the problem here?

```
int * x = new double;
*x = 42;
std::cout << *x << '\n';
```

For

```
int * p = new int;
*p = 42;
```

where is this int value that p points to???

The free store or memory heap

Our variables up to this point (except for pointer variables) have this behavior:

- When you declare it in a block, the variable is created, the memory for the value is also created, and you have access to its name.
- When you exit the block where the variable is declared, the variable name and its value are destroyed.

Note in particular that the scope and value of the variable goes hand in hand. (Remember: The scope of a variable is where you can access the variable's name.) Such variables are called **automatic variables**. Because the memory used for their variables are destroyed and reclaimed automatically when the variable goes out of scope – you don't have to manipulate the memory.

```
if (i > 0)
{
    int x = 42;
    ...
}
// x dies when x goes out of scope. The memory
// used by x is reclaimed automatically by the
// the program.
```

You can actually do this:

```
auto int x = 42;
```

but that's redundant ... C/C++ variables are automatic by default.

When you allocate memory for pointers, the values they point to are different. For instance you have already seen that

$$int * p = new int;$$

creates an `int` value for `p` to point to. Note that

- **the value of `p` (an address) is still automatic ... but**
- **the value `p` points to (an `int`) is not automatic**.

```
if (i > 0)
{
    int * p = new int;
    ...
```
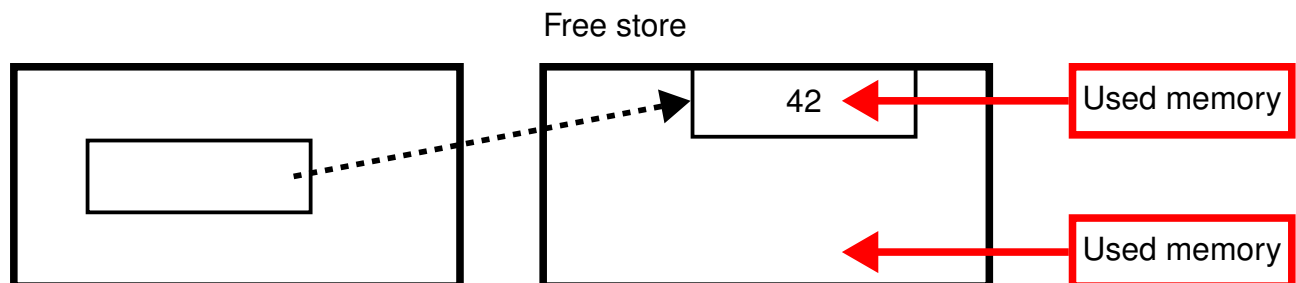
```
}

// p dies when p goes out of scope. The memory
// used by p is reclaimed automatically by the
// program. BUT the value p was pointing to is not
// reclaimed!!!
```
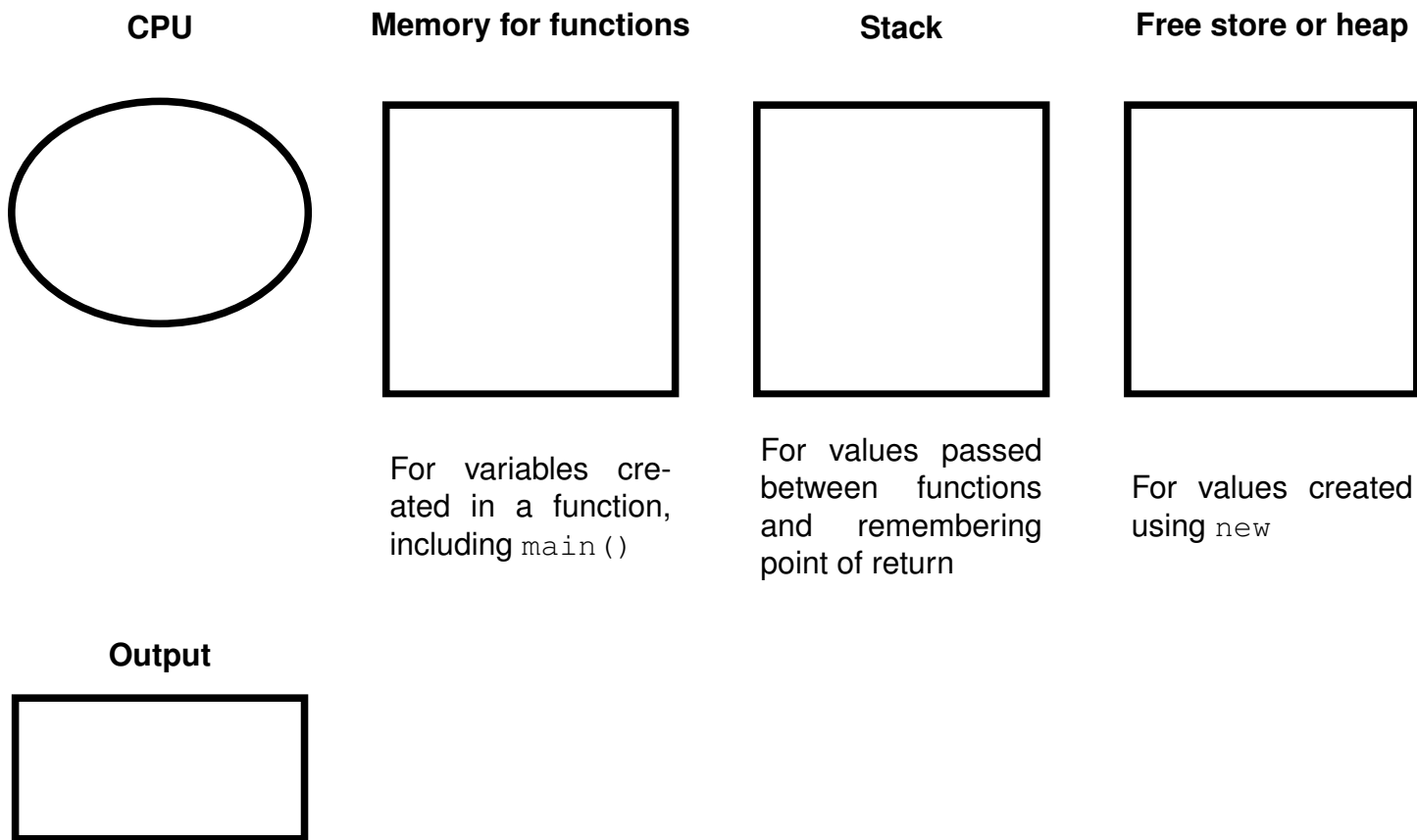
This is **extremely important!!!**

This means that: When you leave the block where you declare a pointer variable `p` and called `new`,

- Since `p` is automatic: the pointer `p` is destroyed and the memory used for the address in `p` reclaimed, but
- Since `*p` is not automatic: the memory that `p` points to is not reclaimed by the computer. We'll talk about how to reclaim the memory later.

But anyway where exactly is the memory `p` points to? It's an an area called the **free store**, also called the **memory heap**. You basically request for memory space using the `new` command.

So our computational model now has the following pieces:

CPU

Memory for functions

Stack

Free store or heap

For variables created in a function, including `main()`

For values passed between functions and remembering point of return

For values created using `new`

Output

The free store will **keep track of which part of its memory is already in use** via `new`. A piece of memory allocated to a pointer variable will be marked as used. That way, **two calls to `new` will not give two pointers the same piece of memory**.

char * p = new char;

int * q = new int;

double * r = new double;

Memory Leaks

I already said that memory allocated via `new` is **not** released back to the free store automatically. **This is a very important point!!!**
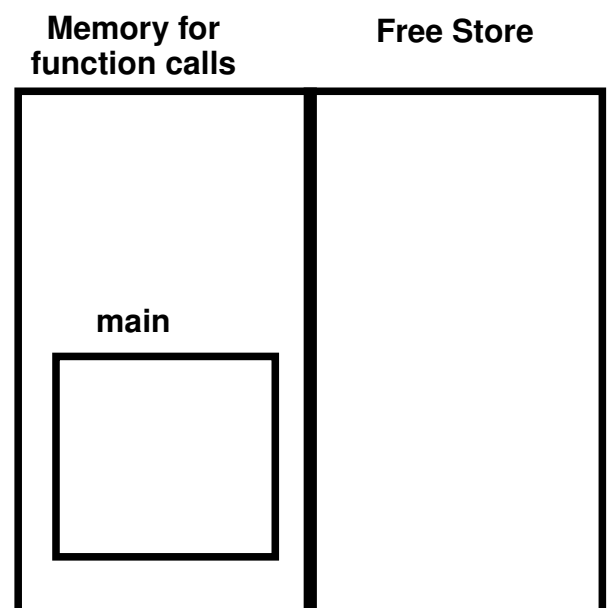
What will happen after calling the following `f` five times?
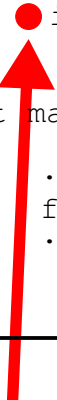
```
void f()
{
    int * p = new int;
}

int main()
{
    ...
    f(); f(); f(); f(); f();
    ...
}
```
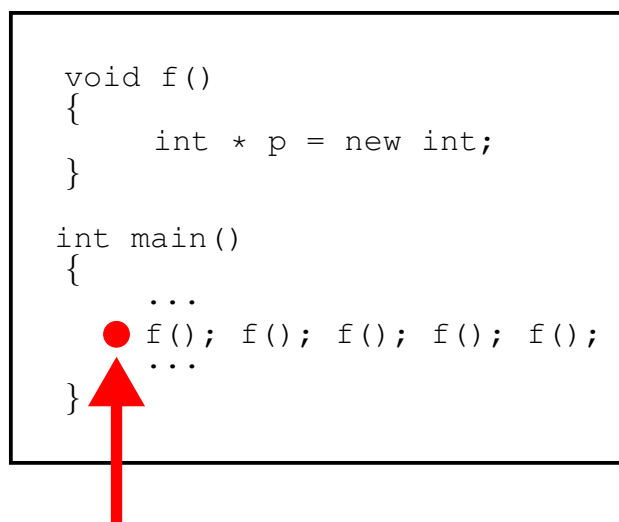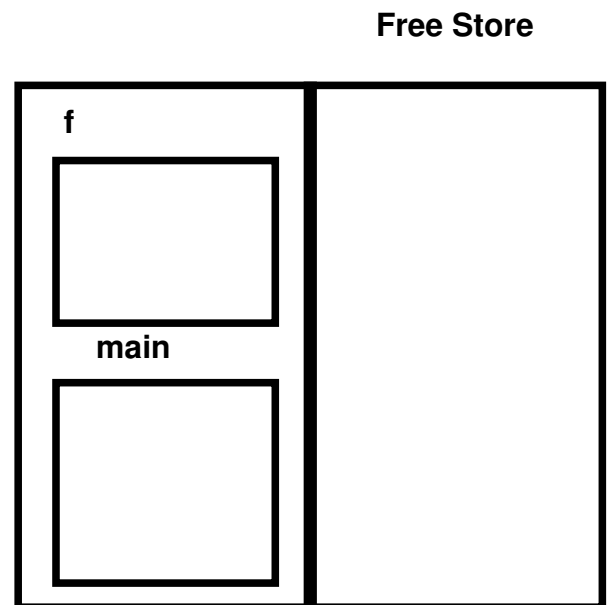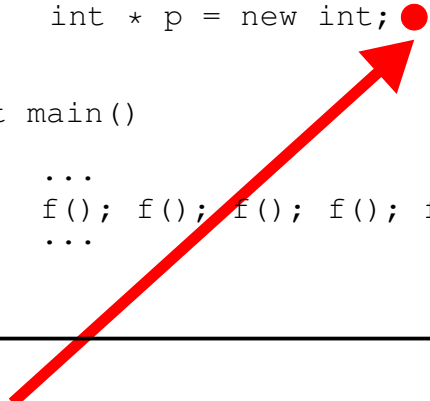
```
void f()
{
    int * p = new int;
}

int main()
{
    ...
  ● f(); f(); f(); f(); f();
    ...
}
```

Program execution at this point

Memory for function calls

Free Store

main

```
void f()
{
    ● int * p = new int;
}

int main()
{
    ...
    f(); f(); f(); f(); f();
    ...
}
```
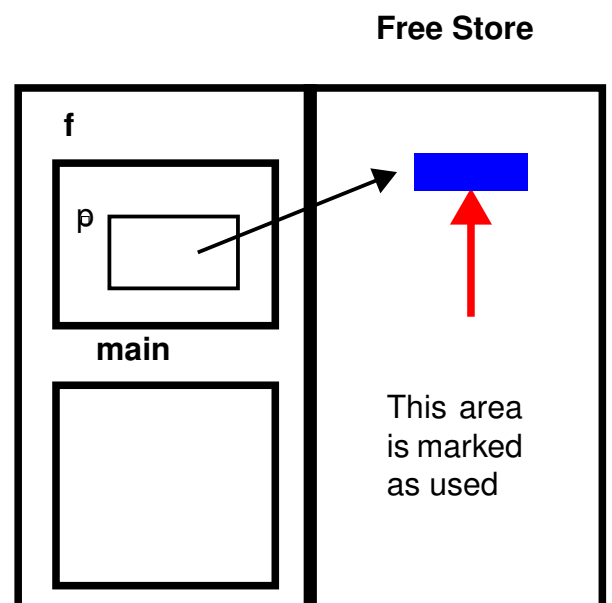
Free Store

f

main

**Program execution at this point
(first time in f)**

```
void f()
{
    int * p = new int; ●
}

int main()
{
    ...
    f(); f(); f(); f(); f();
    ...
}
```

Free Store

f

p

main

This area
is marked
as used

**Program execution at this point
(first time in f)**

Free Store

```
void f()
{
    int * p = new int;
}

int main()
{
    ...
    f(); f(); f(); f(); f();
    ...
}
```

f

p

main

On returning, memory used by `f` is destroyed

Program execution at this point (first time in f, about to return)

Free Store

```
void f()
{
    int * p = new int;
}

int main()
{
    ...
    f(); f(); f(); f(); f();
    ...
}
```

main

this area is still marked as used

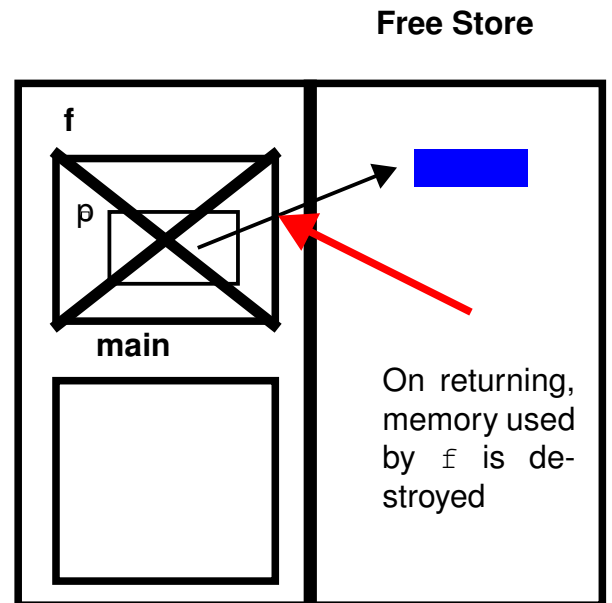**Program execution at this point (about to call f() a second time)**
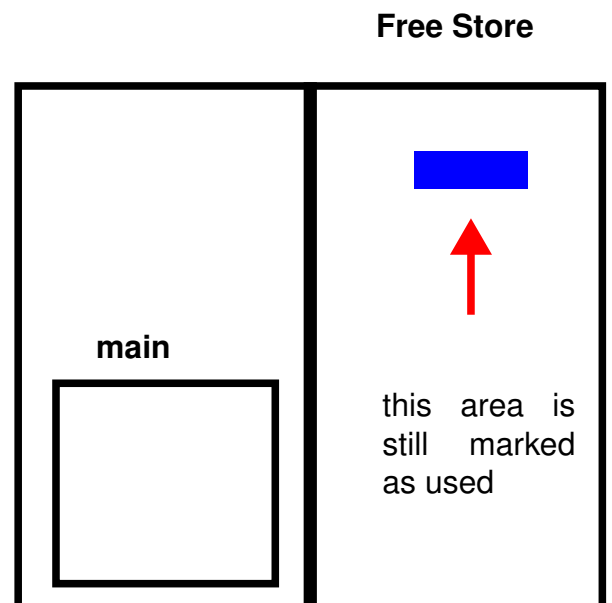
Free Store

```
void f()
{
   ● int * p = new int;
}

int main()
{
   ...
   f(); f(); f(); f(); f();
   ...
}
```

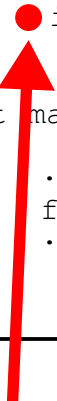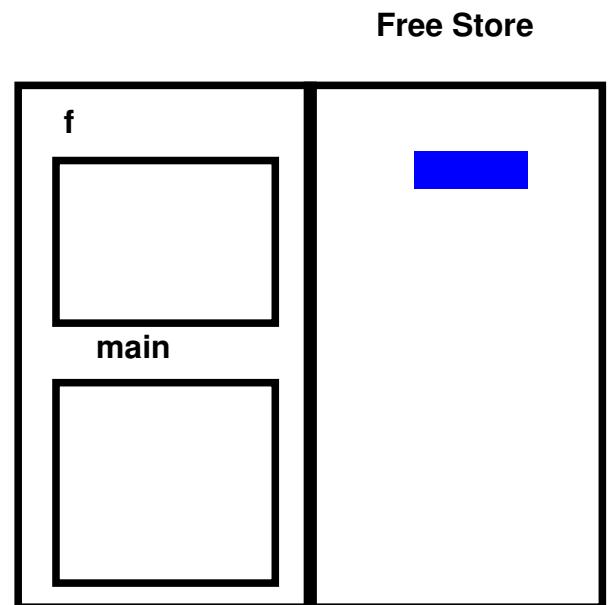**Program execution at this point (about to call f() a second time)**

After some time ... The problem ...

Memory allocated is not released back to free store after use!!! When this happens too frequently, you will run out of memory. Your program might crash!!! The above problem is called a **memory leak** – memory allocated was not deallocated.

Exercise -1.0.66. Run this program.

```
int main()
{
    while (1)
    {
        int * p = new int;
    }
    return 0;
}
```

What will happen after a couple of minutes? Why?

Again ... memory allocated from the free store is **not** automatically reclaimed/released when you exit the block where the memory was allocated. The integer `p` of `f()` is pointing to is not released.

What should we do then??? . . .

delete

After doing a `new`, you can return the memory back to the heap by doing a `delete`. For instance:

```
int * p = new int;

// ... do something here with *p ...

delete p;
```

After `delete p`, the heap will **mark the memory that p is pointing to as available** (i.e. not in use). This is called **deallocating** the memory that `p` is using or **releasing** the memory `p` is using back to the heap.

Here are some really important points:

- Delete pointers ASAP when you don't need them anymore.
- As much as possible, delete pointers in the same block where you new'ed them. You should see new and delete for a pointer in the same block. (There are exceptions.)

After a pointer is deleted it should not be dereferenced. If you need memory again, you just do another `new`:

```
if (x > 42)
{
    int * p = new int;

    // ... do something with *p ...

    delete p;

    // DO NOT USE *p HERE

    p = new int;

    // ... do something with *p ...

    delete p;

    // DO NOT USE *p HERE
}

// p is destroyed
```

Note that when `p` is pointing to memory not in the heap obtained by `new`, then you do NOT `delete p`. For instance:

```
int x = 42;

int * p = &42;

int * q = new int;

delete p; // BAD!!!

delete q; // OK
```

And of course you do not release if you did not `new` it:

```
double * p;

delete p; // Huh????
```

Put in a different way: you cannot give a memory back to the heap if you did not get it from the heap to begin with.

The tendency is to forget to deallocate and then re-allocate, like this:

```
int * p;

// I need an int ...

p = new int;

// ... now use *p

// 1000 lines later ...

// Lack of sleep ... I need an int ...

p = new int;

// ... now use *p

// 1000 lines later ...

// Amnesia kicks in ... I need an int ...

p = new int;

// ... now use *p
```

Can you say "memory leak"?!?! So deallocate ASAP!!! Don't argue!!!

Exercise -1.0.67. The following program does compile and does run. But it has a problem. Fix it by adding ONE statement.

```cpp
#include <iostream>

int sum(int n)
{
    int s = 0;
    int * i = new int;
    for (*i = 0; *i <= n; ++(*i))
    {
        s += *i;
    }
    return s;
}

int main()
{
    std::cout << sum(10) << '\n';
    return 0;
}
```

Exercise -1.0.68. Fixit time ... (you are only allowed to delete one statement.)

```cpp
#include <iostream>

void average(double x[])
{
    double x = 0;
    double * px = &x;
    for (int i = 0; i < 10; ++i)
    {
        *px += x[i];
    }

    x = *px / 10;
    delete px;
    return x;
}

int main()
{
    double x[5] = {1.1, 2.1, 2.3, 3.2, 3.4};
    std::cout << average(x) << std::endl;
    return 0;
}
```

Exercise -1.0.69. The following program prompts the user for two integer values and then prints the sum. Do NOT use integer or double variables. You can only use pointers. In fact I have already declared all the variables you need, i.e., two pointer variables. You must allocate and deallocate memory correctly.

```
#include <iostream>

int main()
{
    int * p;
    int * q;

    // allocate memory for p

    // allocate memory for q

    // prompt for integer value and store at integer

    // that p points to

    // prompt for integer value and store at integer

    // that q points to

    // print the sum of integers that p and q point

    // to

    // deallocate memory used by q

    // deallocate memory used by p

    return 0;
}
```

Exercise -1.0.70. Write a number guessing game. First generate a random integer in the range 1..10. Then prompt the user for an integer value. If his/her guess is correct, print "you got it!". If his/her guess is too low, print "too low!" and prompt him/her again. If his/her guess is too high, print "too high!" and prompt him/her again. OK . . . here's the point, you can only use pointer variables and you must allocate and deallocate memory for your pointers correctly.

Exercise -1.0.71. Deallocate memory appropriately.

```
int f(int a)
```

```
{
    int x = a;
    int * y = &a;
    x = a + *y;
    return 2 * x;
}

int g(int a)
{
    int * y = new int;
    if (a < 0)
    {
        int x = f(2 * a);
        int * y = new int;
        y = a + 1;
        x = 2 * *y;
        return x;
    }
    *y = a + 1;
    a = 2 * a;
    return a;
}

int main()
{
    std::cout << g(42) << std::endl;
    return 0;
}
```

Exercise -1.0.72. Deallocate memory appropriately.

```
int f(int a)
{
    int x = a;
    int * y = &a;
    x = a + *y;
    return 2 * x;
}

int g(int a)
{

    int * y = new int;
    if (a < 0)
    {
        int x = f(2 * a);
        int * y = new int;
        y = a + 1;
        x = 2 * *y;
        return x;
```

```
    }
    *y = a + 1;
    a = 2 * a;
    return a;
}

int main()
{
    std::cout << g(42) << std::endl;
    return 0;
}
```

Use of `NULL` for memory management

Recall that `NULL` is a predefined constant for 0 (as an address) and it's used to denote an invalid address. We can use `NULL` to denote the fact that a pointer has not been allocated.

Look at this:

```
int * p = NULL;
// *p is not used yet

...

// Now you're ready to use an int on the heap
p = new int;
// use *p

...
// Now you don't need the int on the heap anymore
delete p;
p = NULL;

...
```

If you always set your pointer `p` to `NULL` when memory is not allocated for `p` to point to, then you can always check your pointer against `NULL` like this:

```
int * p = NULL;

...

if (p == NULL)
{
    p = new int;
}

// use *p ...
delete p;
p = NULL;

...

if (p == NULL)
{
    p = new int;
}

// use *p ...
delete p;
```

```
p = NULL;

...
```

Why use the free store?

You might ask ... why bother requesting for an `int` inside the free store?

```
int * p = new int;
// ... now use *p
delete p;
```

That's so much trouble ... why not just create a plain `int` variable:

```
int x;
// ... now use x
```

That is true!

Actually the notes from the previous section is just to show you how to use the memory heap. In most cases, you use the heap by requesting for a **very huge** chunk of memory from the free store and not for a single `int`. In other words, dynamic memory management is useful only for large memory usage. As an example, consider the following scenario:

Suppose you have some type `T` that requires a huge amount of memory. Say a variable of type `T` is some kind of an image for a game and a variable of type `T` holds one megabytes of pixel data. Maybe you need 3 such variables:

```
            T bigvar0, bigvar1, bigvar2;
```

In this case all these 3 huge variables will consume 3 MB of memory.

But what if you don't always need them at the same time? Say ...

```
T bigvar0, bigvar1, bigvar2;

// use bigvar0 for 10 min

// use bigvar1 for 10 min

// use bigvar1 and bigvar2 for 10 min

// use bigvar0 for 10 min
```

Then you might want to do this instead:

```
T *bigvar0, *bigvar1, *bigvar2;

bigvar0 = new T; // 1 MB used
// use bigvar0 for 10 min
delete bigvar0; // 0 MB used

bigvar1 = new T; // 1 MB used
// use bigvar1 for 10 min

bigvar2 = new T; // 2 MB used
// use bigvar1 and bigvar2 for 10 min
delete bigvar1; // 1 MB used
delete bigvar2; // 0 MB used

bigvar0 = new T; // 1 MB used
// use bigvar0 for 10 min
delete bigvar0; // 0 MB used
```

This means that you actually only need about 2 MB of memory at any point in time while running the program, not 3 MB.

Get it?

Later I'll talk about requesting not just memory for a value, but a whole array of values. When the array size is huge, this will take up a lot or memory even if each value in the array is only an `int`. For instance:

```
int x[1000000];
```

takes up 4 x 1,000,000 = 4,000,000 bytes. Dynamic memory management is also important when it comes to managing objects (see CISS245, CISS350) since objects usually occupy more memory.

Pointer to Pointer to Pointer to . . .

Now take a **deep** breath . . .

Look at this:

```
int x = 42;
int * p = &x;
```

Here's the memory model:

So `p` has the memory address of some value (in this case an `int` value). But wait a minute. . . memory address are also values . . . so the value of `p` is also somewhere in memory too. For instance suppose the address of the value of x is 124500 and that value is stored in `p` and the value of `p` is stored in 124504:

Since 124504 is also an address, surely you can store 124504 into a variable too. In fact, yes, you can. Looking at this:

```
int x = 42;
int * p = &x;
int ** q = &p;
```

While we say the type of `p` is "pointer to `int`", for `q` we say the type is "pointer to pointer to `int`". Suppose the address of the value of `q` is 124508. Here's the memory model

And here's the simplified picture:

Now think about this: What is `*q`? Using `*p` as a guide,

- `p` points to `x` therefore `*p` is the value of `x`, i.e., `*p is` 42,
- `q` points to `p` therefore `*q` is the value of `p`

So `*q` must be the memory address value stored in pointer variable `p`.

```
int x = 42;
int * p = &x;
int ** q = &p;

std::cout << x << ' ' << *p << '\n';

std::cout << p << ' ' << *q << '\n';
```

But wait ... since `p` is `*q`, you can dereference `p`. . . so you should be

able to dereference *q and get ... right?

- p points to x therefore *p is x
- q points to p therefore *q is p and **q is *p which is x

```
int x = 42;
int * p = &x;
int ** q = &p;

std::cout << x << ' ' << *p << '\n';

std::cout << p << ' ' << *q << '\n';

std::cout << x << ' ' << *p << ' '
          << **q << '\n';
```

Thinking of the dereferencing operator * as "follow the arrow and go into a box", you can think of ** as following **two** arrows and peek into the box you arrive at.

Correct?

To understand the above better, you can go all the way into the actual memory. For instance suppose the value of x lives at address 124500. Remember that an int occupies 4 bytes (for 32 bit machines). And in the computer's memory, the 42 is stored as a bunch of 0s and 1s. But for simplicity, let's just ignore the 0s and 1s of 42 and conceptually put 42 into the memory:

I'm putting x next to 124500 just to remind us that the address of x is 124500. Next, p has the address of x. Memory address also takes up 4 bytes (again assuming we're using a 32-bit machines). Suppose the address of the value of p 124504:

Now it becomes really clear that even p has an address: the value of p (which is a memory address value) sits at address 124504, i.e., &p is 124504. Right? Suppose the address of the value of q, is 124508. The memory looks like this;

Get it? Make sure you see that

- &p is 124504, p is 124500, *p is 42
- &q is 124508, q is 124504, *q is 124500

You see that in some sense knowing low level details actually helps you understand completely what pointers really are and you can see

pointer to pointer very easily. (Or even pointer to pointer to pointer to pointer!)

Of course the three type `int`, `int*`, `int**` are different and obviously you should not assign them to each other (with the exception of 0 which can be assigned to any pointer). So the following is WRONG:

```
int x = 42;

int * y = x;    // ABOMINABLE ... giving int to int*

int ** z = x;   // HORRORS ... giving int to int**

z = y;          // UNSPEAKABLE ... giving int* to int**
```

Exercise -1.0.73. What is the type of `p`?

Write a statement that adds `0.5` to the `3.14` value in the picture using only `p`.

Exercise -1.0.74. At one point in the execution of a program, you have the following memory model:

Write one statement that will change the value of `**p` by incrementing it with the value of `**q`. Use `p` and `q` and not hard-coded constants.

Exercise -1.0.75. At one point in the execution of a program, you have the following memory model:

Is it possible to execute one or more statements to get the following:

If it is, write the statements (use the least number). If it isn't explain why.

Exercise -1.0.76. At one point in the execution of a program, you have the following memory model:

Is it possible to execute one or more statements to get the following:

If it is, write the statements (use the least number please). If it isn't explain why.

Exercise -1.0.77. At one point in the execution of a program, you have the following memory model:

Is it possible to execute one or more statements to get the following:

If it is, write the statements (use the least number please). If it isn't explain why.

Exercise -1.0.78. What is the output? (Or is it an error?)

```
int i = 42;
int * p = &i;
int ** q = &p;

std::cout << *p << ' ' << **q << '\n';
```

(Hint: Draw!!!)

Exercise -1.0.79. What is the output? (Or is it an error?)

```
int i = 42;
int * p = &i;
int ** q = &p;
int *** r = &q;

std::cout << *p << ' ' << **q << ' '
          << ***r << '\n';
```

Exercise -1.0.80. What is the output? (Or is it an error?)

```
int i = 42;
int * p = &i;
int ** q = &p;
int *** r = &q;
***r = 0;

std::cout << *p << ' ' << **q << ' '
          << ***r << '\n';
```

Exercise -1.0.81. What is the output? (Or is it an error?)

```
int i = 42;
int * p = &i;
int * q = *&p;

std::cout << *p << ' ' << *q << '\n';
```

Exercise -1.0.82. What is the output? (Or is it an error?)

```
int i = 42;
int * p = &i;
int *** q = &&p;

std::cout << *p << ' ' << *q << '\n';
```

Exercise -1.0.83. What is the output? (Or is it an error?)

```
int i = 42;
int * p = &i;
int * q = &(*p);

std::cout << *p << ' ' << *q << '\n';
```

Functions that change pointer values

Recall that a function can change a value back in the caller. You can either pass in pointers or use pass-by-reference: Always keep this example in mind:

```cpp
// DOES NOT WORK!!!
// void inc(int x)
// {
// x++;\\
// }

void inc(int * x) // or ``int * const x''
{
    (*x)++;
}

void inc(int & x)
{
    x++;
}

int main()
{
    int x = 42;
    inc(&x);    // x is 43
    inc(x);     // x is 44
}
```

The same idea can be used to change the value of anything you like ...including changing the value of pointers.

Exercise -1.0.84. Explain what's happening in the program below. Is there a problem? How would you fix it?

```cpp
#include <iostream>
#include <cstddef>

void setnull(int * p)
{
    p = NULL;
}

int main()
{
    int * p;
    setnull(p);

    std::cout << (unsigned int) p << std::endl; // 0?
```

```
    return 0;
}
```

Exercise -1.0.85. Explain what's happening in the program below. Is there a problem? How would you fix it?

```cpp
#include <iostream>
#include <cstddef>

void swap(int * p, int * q)
{
    int * t = p;
    p = q;
    q = t;
}

int main()
{
    int x = 0, y = 42;
    int * p = &x, * q = &y;

    std::cout << *p << ' ' << *q << '\n'; // 0 42

    swap(p, q); // p should point to x

                // q should point to y

                // Do you get 42 0?

    std::cout << *p << ' ' << *q << '\n';
    return 0;
}
```

Exercise -1.0.86. Explain what's happening in the program below. Is there a problem?

```cpp
void mynew(int * p)
{
    p = new int;
}

int main()
{
    int * p;
    mynew(p);
    *p = 42;
    return 0;
```

```
}
```

Exercise -1.0.87. Explain what's happening in the program below. Is there a problem?

```
void mydelete(int * p)
{
    delete p;
}

int main()
{
    int * p;
    mydelete(p);
    return 0;
}
```

Exercise -1.0.88. The following function `copy()` copies the data that one pointer is pointing to to the other pointer. In more details, `copy(p, q)` will

- Allocate memory for `p` (if `p` has not been allocated memory; `p` has value `NULL` if it has not been allocated memory)
- Copy the integer `q` points to over to the integer `p` points to.
- Deallocate the memory `q` points to and set `q` to `NULL`.

Test it!