Pointers and Memory

By Nick Parlante

1998-2000, Nick Parlante

Abstract

This document explains how pointers and memory work and how to use them—from the basic concepts through all the major programming techniques. For each topic there is a combination of discussion, sample C code, and drawings.

Audience

This document can be used as an introduction to pointers for someone with basic programming experience. Alternately, it can be used to review and to fill in gaps for someone with a partial understanding of pointers and memory. Many advanced programming and debugging problems only make sense with a complete understanding of pointers and memory — this document tries to provide that understanding. This document concentrates on explaining how pointers work. For more advanced pointer applications and practice problems, see the other resources below.

Pace

Like most CS Education Library documents, the coverage here tries to be complete but fast. The document starts with the basics and advances through all the major topics. The pace is fairly quick — each basic concept is covered once and usually there is some example code and a memory drawing. Then the text moves on to the next topic. For more practice, you can take the time to work through the examples and sample problems. Also, see the references below for more practice problems.

Topics

Topics include: pointers, local memory, allocation, deallocation, dereference operations, pointer assignment, deep vs. shallow copies, the ampersand operator (&), bad pointers, the NULL pointer, value parameters, reference parameters, heap allocation and deallocation, memory ownership models, and memory leaks. The text focuses on pointers and memory in compiled languages like C and C++. At the end of each section, there is some related but optional material, and in particular there are occasional notes on other languages, such as Java.

Pointers and Memory – document #102 in the Stanford CS Education Library. This and other free educational materials are available at http://cslibrary.stanford.edu/102/. This document is free to be used, reproduced, sold, or retransmitted so long as this notice is clearly reproduced at its beginning.

Other CS Education Library Documents

- Point Fun With Binky Video (http://cslibrary.stanford.edu/104/)
  A silly video about pointer basics.

- Linked list Basics (http://cslibrary.stanford.edu/103/)
  Introduces the basic techniques for building linked lists in C.

- Linked List Problems (http://cslibrary.stanford.edu/105/)
  18 classic linked list problems with solutions — a great way to practice
  with realistic, pointer intensive C code, and there's just no substitute for
  practice!

- Essential C (http://cslibrary.stanford.edu/101/)
  Complete coverage of the C language, including all of the syntax used in
  this document.

Table of Contents

Edition

The first edition of this document was on Jan 19, 1999. This Feb 21, 2000 edition
represents only very minor changes. The author may be reached at
nick.parlante@cs.stanford.edu. The CS Education Library may be reached at
cslibrary@cs.stanford.edu.

Dedication

This document is distributed for the benefit and education of all. That someone seeking
education should have the opportunity to find it. May you learn from it in the spirit in
which it is given — to make efficiency and beauty in your designs, peace and fairness in
your actions.

Preface To The First Edition

This article has appeared to hover at around 80% done for 6 months! Every time I add
one section, I think of two more which also need to be written. I was motivated to keep
working on it since there are so many other articles which use memory, &, ... in passing
where I wanted something to refer to. I hope you find it valuable in its current form. I'm
going to ship it quickly before I accidentally start adding another section!

Section 1 — Basic Pointers

Pointers — Before and After

There's a lot of nice, tidy code you can write without knowing about pointers. But once you learn to use the power of pointers, you can never go back. There are too many things that can only be done with pointers. But with increased power comes increased responsibility. Pointers allow new and more ugly types of bugs, and pointer bugs can crash in random ways which makes them more difficult to debug. Nonetheless, even with their problems, pointers are an irresistibly powerful programming construct. (The following explanation uses the C language syntax where a syntax is required; there is a discussion of Java at the section.)

Why Have Pointers?

Pointers solve two common software problems. First, pointers allow different sections of code to share information easily. You can get the same effect by copying information back and forth, but pointers solve the problem better. Second, pointers enable complex "linked" data structures like linked lists and binary trees.
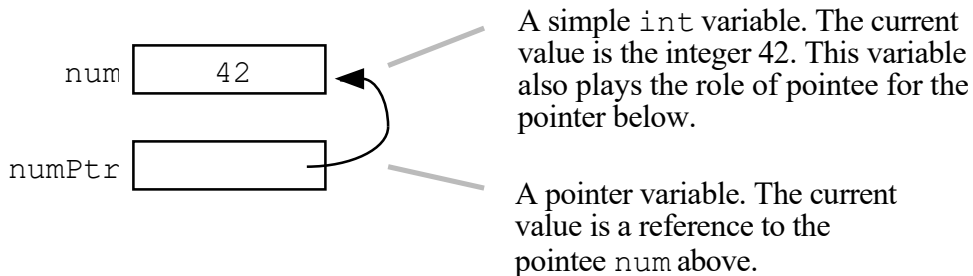
What Is A Pointer?

Simple `int` and `float` variables operate pretty intuitively. An `int` variable is like a box which can store a single `int` value such as 42. In a drawing, a simple variable is a box with its current value drawn inside.

num | 42 |
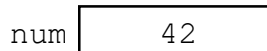
A pointer works a little differently— it does not store a simple value directly. Instead, a pointer stores a **reference** to another value. The variable the pointer refers to is sometimes known as its "pointee". In a drawing, a pointer is a box which contains the beginning of an arrow which leads to its pointee. (There is no single, official, word for the concept of a pointee — pointee is just the word used in these explanations.)

The following drawing shows two variables: `num` and `numPtr`. The simple variable `num` contains the value 42 in the usual way. The variable `numPtr` is a pointer which contains a reference to the variable `num`. The `numPtr` variable is the pointer and `num` is its pointee. What is stored inside of `numPtr`? Its value is not an `int`. Its value is a reference to an `int`.

num | 42 |

A simple `int` variable. The current value is the integer 42. This variable also plays the role of pointee for the pointer below.

numPtr | |

A pointer variable. The current value is a reference to the pointee `num` above.

Pointer Dereference

The "dereference" operation follows a pointer's reference to get the value of its pointee. The value of the dereference of numPtr above is 42. When the dereference operation is used correctly, it's simple. It just accesses the value of the pointee. The only restriction is that the pointer must have a pointee for the dereference to access. Almost all bugs in pointer code involve violating that one restriction. A pointer must be assigned a pointee before dereference operations will work.

The NULL Pointer

The constant NULL is a special pointer value which encodes the idea of "points to nothing." It turns out to be convenient to have a well defined pointer value which represents the idea that a pointer does not have a pointee. It is a runtime error to dereference a NULL pointer. In drawings, the value NULL is usually drawn as a diagonal line between the corners of the pointer variable's box...

The C language uses the symbol NULL for this purpose. NULL is equal to the integer constant 0, so NULL can play the role of a boolean false. Official C++ no longer uses the NULL symbolic constant — use the integer constant 0 directly. Java uses the symbol null.

Pointer Assignment

The assignment operation (=) between two pointers makes them point to the same pointee. It's a simple rule for a potentially complex situation, so it is worth repeating: assigning one pointer to another makes them point to the same thing. The example below adds a second pointer, second, assigned with the statement second = numPtr;. The result is that second points to the same pointee as numPtr. In the drawing, this means that the second and numPtr boxes both contain arrows pointing to num. Assignment between pointers does not change or even touch the pointees. It just changes which pointee a pointer refers to.
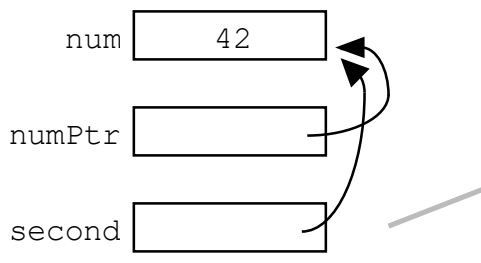
A second pointer ptr initialized with the assignment second = numPtr;. This causes second to refer to the same pointeee as numPtr.

After assignment, the == test comparing the two pointers will return true. For example (second==numPtr) above is true. The assignment operation also works with the NULL value. An assignment operation with a NULL pointer copies the NULL value from one pointer to another.

Make A Drawing

Memory drawings are the key to thinking about pointer code. When you are looking at code, thinking about how it will use memory at run time....make a quick drawing to work out your ideas. This article certainly uses drawings to show how pointers work. That's the way to do it.
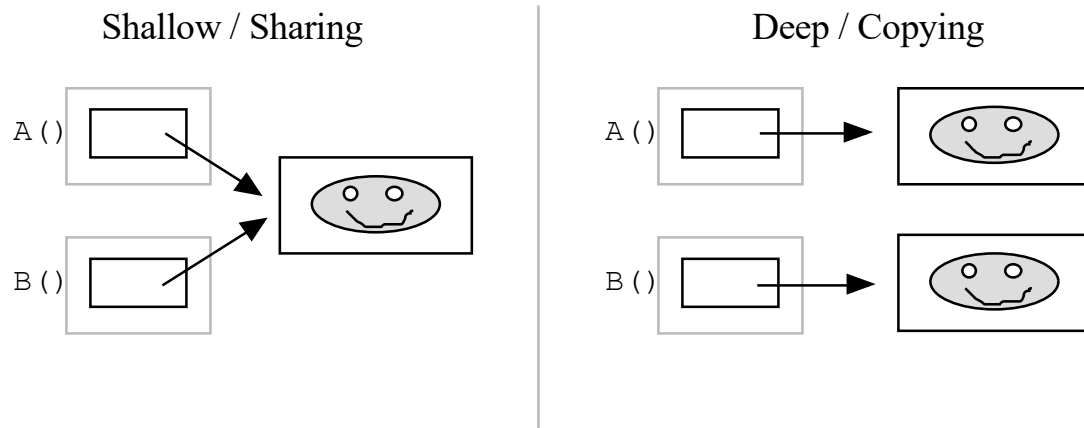
Sharing

Two pointers which both refer to a single pointee are said to be "sharing". That two or more entities can cooperatively share a single memory structure is a key advantage of pointers in all computer languages. Pointer manipulation is just technique — sharing is often the real goal. In Section 3 we will see how sharing can be used to provide efficient communication between parts of a program.

Shallow and Deep Copying

In particular, sharing can enable communication between two functions. One function passes a pointer to the value of interest to another function. Both functions can access the value of interest, but the value of interest itself is not copied. This communication is called "shallow" since instead of making and sending a (large) copy of the value of interest, a (small) pointer is sent and the value of interest is shared. The recipient needs to understand that they have a shallow copy, so they know not to change or delete it since it is shared. The alternative where a complete copy is made and sent is known as a "deep" copy. Deep copies are simpler in a way, since each function can change their copy without interfering with the other copy, but deep copies run slower because of all the copying.

The drawing below shows shallow and deep copying between two functions, A() and B(). In the shallow case, the smiley face is shared by passing a pointer between the two. In the deep case, the smiley face is copied, and each function gets their own...

Shallow / Sharing · Deep / Copying

Section 2 will explain the above sharing technique in detail.

Bad Pointers

When a pointer is first allocated, it does not have a pointee. The pointer is "uninitialized" or simply "bad". A dereference operation on a bad pointer is a serious runtime error. If you are lucky, the dereference operation will crash or halt immediately (Java behaves this way). If you are unlucky, the bad pointer dereference will corrupt a random area of memory, slightly altering the operation of the program so that it goes wrong some indefinite time later. Each pointer must be assigned a pointee before it can support dereference operations. Before that, the pointer is bad and must not be used. In our memory drawings, the bad pointer value is shown with an XXX value...

numPtr ☒ ☒ ☒

Bad pointers are very common. In fact, **every pointer starts out with a bad value**. Correct code overwrites the bad value with a correct reference to a pointee, and thereafter the pointer works fine. There is nothing automatic that gives a pointer a valid pointee.

Quite the opposite — most languages make it easy to omit this important step. You just have to program carefully. If your code is crashing, a bad pointer should be your first suspicion.

Pointers in dynamic languages such as Perl, LISP, and Java work a little differently. The run-time system sets each pointer to NULL when it is allocated and checks it each time it is dereferenced. So code can still exhibit pointer bugs, but they will halt politely on the offending line instead of crashing haphazardly like C. As a result, it is much easier to locate and fix pointer bugs in dynamic languages. The run-time checks are also a reason why such languages always run at least a little slower than a compiled language like C or C++.

Two Levels
One way to think about pointer code is that operates at two levels — pointer level and pointee level. The trick is that **both** levels need to be initialized and connected for things to work. (1) the pointer must be allocated, (1) the pointee must be allocated, and (3) the pointer must be assigned to point to the pointee. It's rare to forget step (1). But forget (2) or (3), and the whole thing will blow up at the first dereference. Remember to account for both levels — make a memory drawing during your design to make sure it's right.

Syntax
The above basic features of pointers, pointees, dereferencing, and assigning are the only concepts you need to build pointer code. However, in order to talk about pointer code, we need to use a known syntax which is about as interesting as....a syntax. We will use the C language syntax which has the advantage that it has influenced the syntaxes of several languages.

Pointer Type Syntax
A pointer type in C is just the pointee type followed by a asterisk (*)...

`int*`	type: pointer to `int`
`float*`	type: pointer to `float`
`struct fraction*`	type: pointer to `struct fraction`
`struct fraction**`	type: pointer to `struct fraction*`

Pointer Variables
Pointer variables are declared just like any other variable. The declaration gives the type and name of the new variable and reserves memory to hold its value. The declaration does not assign a pointee for the pointer — the pointer starts out with a bad value.

```
int* numPtr;   // Declare the int* (pointer to int) variable "numPtr".
               // This allocates space for the pointer, but not the pointee.
               // The pointer starts out "bad".
```

The & Operator — Reference To

There are several ways to compute a reference to a pointee suitable for storing in a pointer. The simplest way is the & operator. The & operator can go to the left of any variable, and it computes a reference to that variable. The code below uses a pointer and an & to produce the earlier num/numPtr example.

```
num [    42    ]  ◀
                    ╲
numPtr [        ]──╯
```
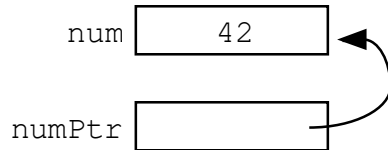
```
void NumPtrExample() {
   int num;
   int* numPtr;

   num = 42;
   numPtr = &num; // Compute a reference to "num", and store it in numPtr
   // At this point, memory looks like drawing above
}
```

It is possible to use & in a way which compiles fine but which creates problems at run time — the full discussion of how to correctly use & is in Section 2. For now we will just use & in a simple way.

The * Operator — Dereference

The star operator (*) dereferences a pointer. The * is a unary operator which goes to the left of the pointer it dereferences. The pointer must have a pointee, or it's a runtime error.

Example Pointer Code

With the syntax defined, we can now write some pointer code that demonstrates all the pointer rules...
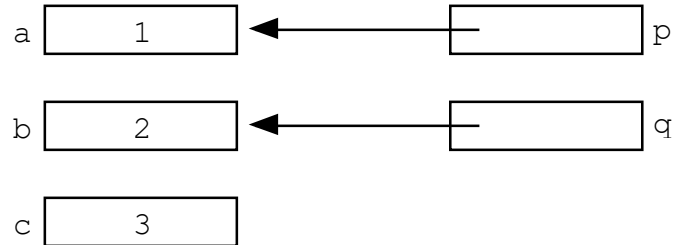
```
void PointerTest() {
   // allocate three integers and two pointers
   int a = 1;
   int b = 2;
   int c = 3;
   int* p;
   int* q;

   // Here is the state of memory at this point.
   // T1 -- Notice that the pointers start out bad...
```
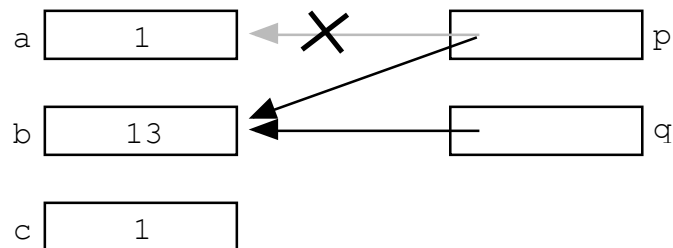
```
a [    1    ]          [ ✕ ✕ ✕ ] p

b [    2    ]          [ ✕ ✕ ✕ ] q

c [    3    ]
```
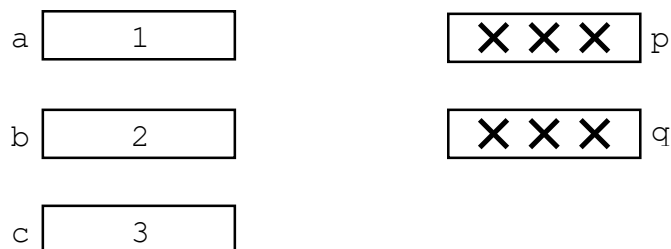
```
   p = &a;     // set p to refer to a
```

```
    q = &b;      // set q to refer to b
    // T2 -- The pointers now have pointees
```

```
a [    1    ]  ◄——————  [        ]  p

b [    2    ]  ◄——————  [        ]  q

c [    3    ]
```

```
    // Now we mix things up a bit...
    c = *p;  // retrieve p's pointee value (1) and put it in c
    p = q;   // change p to share with q (p's pointee is now b)
    *p = 13; // dereference p to set its pointee (b) to 13 (*q is now 13)
    // T3 -- Dereferences and assignments mix things up
```

```
a [    1    ]  ◄——✗——  [        ]  p

b [   13    ]  ◄——————  [        ]  q

c [    1    ]
```

```
}
```

Bad Pointer Example

Code with the most common sort of pointer bug will look like the above correct code, but without the middle step where the pointers are assigned pointees. The bad code will compile fine, but at run-time, each dereference with a bad pointer will corrupt memory in some way. The program will crash sooner or later. It is up to the programmer to ensure that each pointer is assigned a pointee before it is used. The following example shows a simple example of the bad code and a drawing of how memory is likely to react...

```
void BadPointer() {
    int* p;      // allocate the pointer, but not the pointee

    *p = 42;     // this dereference is a serious runtime error
}
// What happens at runtime when the bad pointer is dereferenced...
```

```
p [ ✗ ✗ ✗ ]——~~~~~►  Pow!
```
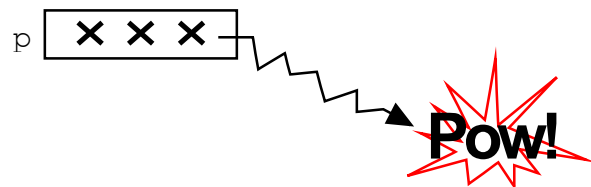
Pointer Rules Summary

No matter how complex a pointer structure gets, the list of rules remains short.

- A pointer stores a reference to its pointee. The pointee, in turn, stores something useful.

- The dereference operation on a pointer accesses its pointee. A pointer may only be dereferenced after it has been assigned to refer to a pointee. Most pointer bugs involve violating this one rule.

- Allocating a pointer does not automatically assign it to refer to a pointee. Assigning the pointer to refer to a specific pointee is a separate operation which is easy to forget.

- Assignment between two pointers makes them refer to the same pointee which introduces sharing.

Section 1 — Extra Optional Material

Extra: How Do Pointers Work In Java

Java has pointers, but they are not manipulated with explicit operators such as * and &. In Java, simple data types such as `int` and `char` operate just as in C. More complex types such as arrays and objects are automatically implemented using pointers. The language automatically uses pointers behind the scenes for such complex types, and no pointer specific syntax is required. The programmer just needs to realize that operations like `a=b;` will automatically be implemented with pointers if a and b are arrays or objects. Or put another way, the programmer needs to remember that assignments and parameters with arrays and objects are intrinsically shallow or shared— see the Deep vs. Shallow material above. The following code shows some Java object references. Notice that there are no *'s or &'s in the code to create pointers. The code intrinsically uses pointers. Also, the garbage collector (Section 4), takes care of the deallocation automatically at the end of the function.

```
public void JavaShallow() {
    Foo a = new Foo();   // Create a Foo object (no * in the declaration)
    Foo b = new Foo();   // Create another Foo object

    b=a;  // This is automatically a shallow assignment --
          // a and b now refer to the same object.

    a.Bar(); // This could just as well be written b.Bar();

    // There is no memory leak here -- the garbage collector
    // will automatically recycle the memory for the two objects.
}
```

The Java approach has two main features...

- *Fewer bugs*. Because the language implements the pointer manipulation accurately and automatically, the most common pointer bug are no longer possible, Yay! Also, the Java runtime system checks each pointer value every time it is used, so NULL pointer dereferences are caught immediately on the line where they occur. This can make a programmer much more productive.

- *Slower*. Because the language takes responsibility for implementing so much pointer machinery at runtime, Java code runs slower than the equivalent C code. (There are other reasons for Java to run slowly as well. There is active research in making Java faser in interesting ways — the Sun "Hot Spot" project.) In any case, the appeal of increased programmer efficiency and fewer bugs makes the slowness worthwhile for some applications.

Extra: How Are Pointers Implemented In The Machine?

How are pointers implemented? The short explanation is that every area of memory in the machine has a numeric address like 1000 or 20452. A pointer to an area of memory is really just an integer which is storing the address of that area of memory. The dereference operation looks at the address, and goes to that area of memory to retrieve the pointee stored there. Pointer assignment just copies the numeric address from one pointer to another. The NULL value is generally just the numeric address 0 — the computer just never allocates a pointee at 0 so that address can be used to represent NULL. A bad pointer is really just a pointer which contains a random address — just like an uninitialized `int` variable which starts out with a random `int` value. The pointer has not yet been assigned the specific address of a valid pointee. This is why dereference operations with bad pointers are so unpredictable. They operate on whatever random area of memory they happen to have the address of.

Extra: The Term "Reference"

The word "reference" means almost the same thing as the word "pointer". The difference is that "reference" tends to be used in a discussion of pointer issues which is not specific to any particular language or implementation. The word "pointer" connotes the common C/C++ implementation of pointers as addresses. The word "reference" is also used in the phrase "reference parameter" which is a technique which uses pointer parameters for two-way communication between functions — this technique is the subject of Section 3.

Extra: Why Are Bad Pointer Bugs So Common?

Why is it so often the case that programmers will allocate a pointer, but forget to set it to refer to a pointee? The rules for pointers don't seem that complex, yet every programmer makes this error repeatedly. Why? The problem is that we are trained by the tools we use. Simple variables don't require any extra setup. You can allocate a simple variable, such as `int`, and use it immediately. All that `int`, `char`, `struct fraction` code you have written has trained you, quite reasonably, that a variable may be used once it is declared. Unfortunately, pointers **look** like simple variables but they require the extra initialization before use. It's unfortunate, in a way, that pointers happen look like other variables, since it makes it easy to forget that the rules for their use are very different. Oh well. Try to remember to assign your pointers to refer to pointees. Don't be surprised when you forget.

Section 2 —
Local Memory

Thanks For The Memory

Local variables are the programming structure everyone uses but no one thinks about. You think about them a little when first mastering the syntax. But after a few weeks, the variables are so automatic that you soon forget to think about how they work. This situation is a credit to modern programming languages— most of the time variables appear automatically when you need them, and they disappear automatically when you are finished. For basic programming, this is a fine situation. However, for advanced programming, it's going to be useful to have an idea of how variables work...

Allocation And Deallocation

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

The most common memory related error is using a deallocated variable. For local variables, modern languages automatically protect against this error. With pointers, as we will see however, the programmer must make sure that allocation is handled correctly..

Local Memory

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

```
// Local storage example
int Square(int num) {
   int result;

   result = num * num;

   return result;
}
```

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

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

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

2. The memory for the locals continues to be allocated so long as the thread of control is within the owning function. Locals continue to exist even if the function temporarily passes off the thread of control by calling another function. The locals exist undisturbed through all of this.

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

Small Locals Example
Here is a simple example of the lifetime of local storage...

```
void Foo(int a) { // (1) Locals (a, b, i, scores) allocated when Foo runs
    int i;
    float scores[100];    // This array of 100 floats is allocated locally.

    a = a + 1;  // (2) Local storage is used by the computation
    for (i=0; i<a; i++) {
        Bar(i + a);        // (3) Locals continue to exist undisturbed,
    }                      // even during calls to other functions.


}     // (4) The locals are all deallocated when the function exits.
```

Large Locals Example
Here is a larger example which shows how the simple rule "the locals are allocated when their function begins running and are deallocated when it exits" can build more complex behavior. You will need a firm grasp of how local allocation works to understand the material in sections 3 and 4 later.

The drawing shows the sequence of allocations and deallocations which result when the function X() calls the function Y() twice. The points in time T1, T2, etc. are marked in the code and the state of memory at that time is shown in the drawing.
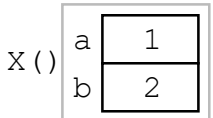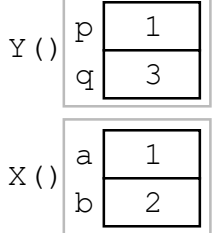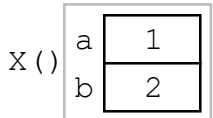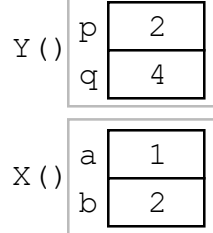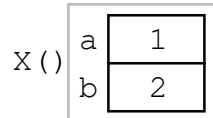
```
void X() {
   int a = 1;
   int b = 2;
   // T1

   Y(a);
   // T3
   Y(b);

   // T5


}

void Y(int p) {
   int q;
   q = p + 2;
   // T2 (first time through), T4 (second time through)
}
```

T1 - X()'s locals have been allocated and given values..	T2 - Y() is called with p=1, and its locals are allocated. X()'s locals continue to be allocated.	T3 - Y() exits and its locals are deallocated. We are left only with X()'s locals.	T4 - Y() is called again with p=2, and its locals are allocated a second time.	T5 - Y() exits and its locals are deallocated. X()'s locals will be deallocated when it exits.
X() a 1 / b 2	Y() p 1 / q 3 X() a 1 / b 2	X() a 1 / b 2	Y() p 2 / q 4 X() a 1 / b 2	X() a 1 / b 2

(optional extra...) The drawing shows the sequence of the locals being allocated and deallocated — in effect the drawing shows the operation over time of the "stack" which is the data structure which the system uses to implement local storage.

Observations About Local Parameters
Local variables are tightly associated with their function — they are used there and nowhere else. Only the X() code can refer to its a and b. Only the Y() code can refer to its p and q. This independence of local storage is the root cause of both its advantages and disadvantages.

Advantages Of Locals
Locals are great for 90% of a program's memory needs....

Convenient. Locals satisfy a convenient need — functions often need some temporary memory which exists only during the function's computation. Local variables conveniently provide this sort of temporary, independent memory.

Efficient. Relative to other memory use techniques, locals are very efficient. Allocating and deallocating them is time efficient (fast) and they are space efficient in the way they use and recycle memory.

Local Copies. Local parameters are basically local **copies** of the information from the caller. This is also known as "pass by value." Parameters are local variables which are initialized with an assignment (=) operation from the caller. The caller is not "sharing" the parameter value with the callee in the pointer sense— the callee is getting its own copy. This has the advantage that the callee can change its local copy without affecting the caller. (Such as with the "p" parameter in the above example.) This independence is good since it keeps the operation of the caller and callee functions separate which follows the rules of good software engineering — keep separate components as independent as possible.

Disadvantages Of Locals
There are two disadvantages of Locals

Short Lifetime. Their allocation and deallocation schedule (their "lifetime") is very strict. Sometimes a program needs memory which continues to be allocated even after the function which originally allocated it has exited. Local variables will not work since they are deallocated automatically when their owning function exits. This problem will be solved later in Section 4 with "heap" memory.

Restricted Communication. Since locals are copies of the caller parameters, they do not provide a means of communication from the callee back to the caller. This is the downside of the "independence" advantage. Also, sometimes making copies of a value is undesirable for other reasons. We will see the solution to this problem below in Section 3 "Reference Parameters".

Synonyms For "Local"
Local variables are also known as "automatic" variables since their allocation and deallocation is done automatically as part of the function call mechanism. Local variables are also sometimes known as "stack" variables because, at a low level, languages almost always implement local variables using a stack structure in memory.

The Ampersand (&) Bug — TAB
Now that you understand the allocation schedule of locals, you can appreciate one of the more ugly bugs possible in C and C++. What is wrong with the following code where the function Victim() calls the function TAB()? To see the problem, it may be useful to make a drawing to trace the local storage of the two functions...

```
// TAB -- The Ampersand Bug function
// Returns a pointer to an int
int* TAB() {
   int temp;
   return(&temp); // return a pointer to the local int
}

void Victim() {
   int* ptr;
   ptr = TAB();
   *ptr = 42;      // Runtime error! The pointee was local to TAB
}
```

TAB() is actually fine while it is running. The problem happens to its caller after TAB() exits. TAB() returns a pointer to an `int`, but where is that `int` allocated? The problem is that the local `int`, `temp`, is allocated only while TAB() is running. When TAB() exits, all of its locals are deallocated. So the caller is left with a pointer to a deallocated variable. TAB()'s locals are deallocated when it exits, just as happened to the locals for Y() in the previous example.

It is incorrect (and useless) for TAB() to return a pointer to memory which is about to be deallocated. We are essentially running into the "lifetime" constraint of local variables. We want the int to exist, but it gets deallocated automatically. Not all uses of & between functions are incorrect — only when used to pass a pointer back to the caller. The correct uses of & are discussed in section 3, and the way to pass a pointer back to the caller is shown in section 4.

Local Memory Summary
Locals are very convenient for what they do — providing convenient and efficient memory for a function which exists only so long as the function is executing. Locals have two deficiencies which we will address in the following sections — how a function can communicate back to its caller (Section 3), and how a function can allocate separate memory with a less constrained lifetime (section 4).

Section 2 — Extra Optional Material

Extra: How Does The Function Call Stack Work?
You do not need to know how local variables are implemented during a function call, but here is a rough outline of the steps if you are curious. The exact details of the implementation are language and compiler specific. However, the basic structure below is approximates the method used by many different systems and languages...

To call a function such as `foo(6, x+1)`...

1. Evaluate the actual parameter expressions, such as the x+1, in the caller's context.

2. Allocate memory for foo()'s locals by pushing a suitable "local block" of memory onto a runtime "call stack" dedicated to this purpose. For parameters but not local variables, store the values from step (1) into the appropriate slot in foo()'s local block.

3. Store the caller's current address of execution (its "return address") and switch execution to foo().

4. foo() executes with its local block conveniently available at the end of the call stack.

5. When foo() is finished, it exits by popping its locals off the stack and "returns" to the caller using the previously stored return address. Now the caller's locals are on the end of the stack and it can resume executing.

For the extremely curious, here are other miscellaneous notes on the function call process...

- This is why infinite recursion results in a "Stack Overflow Error" — the code keeps calling and calling resulting in steps (1) (2) (3), (1) (2) (3), but never a step (4)....eventually the call stack runs out of memory.

- This is why local variables have random initial values — step (2) just pushes the whole local block in one operation. Each local gets its own area of memory, but the memory will contain whatever the most recent tenant left there. To clear all of the local block for each function call would be too time expensive.

- The "local block" is also known as the function's "activation record" or "stack frame". The entire block can be pushed onto the stack (step 2), in a single CPU operation — it is a very fast operation.

- For a multithreaded environment, each thread gets its own call stack instead of just having single, global call stack.

- For performance reasons, some languages pass some parameters through registers and others through the stack, so the overall process is complex. However, the apparent the lifetime of the variables will always follow the "stack" model presented here.

Section 3 —
Reference Parameters

In the simplest "pass by value" or "value parameter" scheme, each function has separate, local memory and parameters are copied from the caller to the callee at the moment of the function call. But what about the other direction? How can the callee communicate back to its caller? Using a "return" at the end of the callee to copy a result back to the caller works for simple cases, but does not work well for all situations. Also, sometimes copying values back and forth is undesirable. "Pass by reference" parameters solve all of these problems.

For the following discussion, the term "value of interest" will be a value that the caller and callee wish to communicate between each other. A reference parameter passes a pointer to the value of interest instead of a copy of the value of interest. This technique uses the sharing property of pointers so that the caller and callee can share the value of interest.

Bill Gates Example
Suppose functions A() and B() both do computations involving Bill Gates' net worth measured in billions of dollars — the value of interest for this problem. A() is the main function and its stores the initial value (about 55 as of 1998). A() calls B() which tries to add 1 to the value of interest.

Bill Gates By Value
Here is the code and memory drawing for a simple, but incorrect implementation where A() and B() use pass by value. Three points in time, T1, T2, and T3 are marked in the code and the state of memory is shown for each state...

```
void B(int worth) {
   worth = worth + 1;
   // T2
}
void A() {
   int netWorth;
   netWorth = 55; // T1

   B(netWorth);
   // T3 -- B() did not change netWorth
}
```
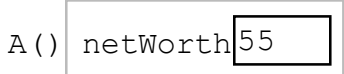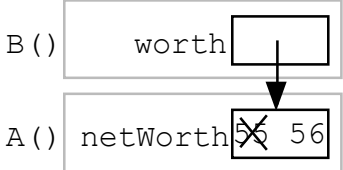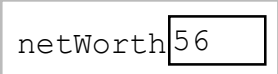
T1 -- The value of interest `netWorth` is local to A().	T2 -- `netWorth` is copied to B()'s local `worth`. B() changes its local worth from 55 to 56.	T3 -- B() exits and its local `worth` is deallocated. The value of interest has not been changed.
	B() worth ⊠ 56	
A() netWorth 55	A() netWorth 55	A() netWorth 55

B() adds 1 to its local `worth` copy, but when B() exits, `worth` is deallocated, so changing it was useless. The value of interest, `netWorth`, rests unchanged the whole time in A()'s local storage. A function can change its local copy of the value of interest, but that change is not reflected back in the original value. This is really just the old "independence" property of local storage, but in this case it is not what is wanted.

By Reference

The reference solution to the Bill Gates problem is to use a single `netWorth` variable for the value of interest and never copy it. Instead, each function can receives a pointer to `netWorth`. Each function can see the current value of `netWorth` by dereferencing its pointer. More importantly, each function can change the net `worth` — just dereference the pointer to the centralized `netWorth` and change it directly. Everyone agrees what the current value of `netWorth` because it exists in only one place — everyone has a pointer to the one master copy. The following memory drawing shows A() and B() functions changed to use "reference" parameters. As before, T1, T2, and T3 correspond to points in the code (below), but you can study the memory structure without looking at the code yet.

T1 -- The value of interest, `netWorth`, is local to A() as before.	**T2** -- Instead of a copy, B() receives a pointer to `netWorth`. B() dereferences its pointer to access and change the real `netWorth`.	**T3** -- B() exits, and `netWorth` has been changed.
A() netWorth 55	B() worth ⟶ A() netWorth ⟩⟨ 56	A() netWorth 56

The reference parameter strategy: B() receives a pointer to the value of interest instead of a copy.

Passing By Reference

Here are the steps to use in the code to use the pass-by-reference strategy...

- Have a single copy of the value of interest. The single "master" copy.

- Pass pointers to that value to any function which wants to see or change the value.

- Functions can dereference their pointer to see or change the value of interest.

- Functions must remember that they do not have their own local copies. If they dereference their pointer and change the value, they really are changing the master value. If a function wants a local copy to change safely, the function must explicitly allocate and initialize such a local copy.

Syntax

The syntax for by reference parameters in the C language just uses pointer operations on the parameters...

1. Suppose a function wants to communicate about some value of interest — int or float or struct fraction.

2. The function takes as its parameter a pointer to the value of interest — an int* or float* or struct fraction*. Some programmers will add the word "ref" to the name of a reference parameter as a reminder that it is a reference to the value of interest instead of a copy.

3. At the time of the call, the caller computes a pointer to the value of interest and passes that pointer. The type of the pointer (pointer to the value of interest) will agree with the type in (2) above. If the value of interest is local to the caller, then this will often involve a use of the & operator (Section 1).

4. When the callee is running, if it wishes to access the value of interest, it must dereference its pointer to access the actual value of interest. Typically, this equates to use of the dereference operator (*) in the function to see the value of interest.

Bill Gates By Reference

Here is the Bill Gates example written to use reference parameters. This code now matches the by-reference memory drawing above.

```
// B() now uses a reference parameter -- a pointer to
// the value of  interest. B() uses a dereference (*) on the
// reference parameter to get at the value of interest.
void B(int* worthRef) {        // reference parameter
   *worthRef = *worthRef + 1; // use * to get at value of interest
   // T2
}

void A() {
   int netWorth;
   netWorth = 55;         // T1 -- the value of interest is local to A()

   B(&netWorth);  // Pass a pointer to the value of interest.
                  // In this case using &.

   // T3 -- B() has used its pointer to change the value of interest
}
```

Don't Make Copies

Reference parameters enable communication between the callee and its caller. Another reason to use reference parameters is to avoid making copies. For efficiency, making copies may be undesirable if the value of interest is large, such as an array. Making the copy requires extra space for the copy itself and extra time to do the copying. From a design point of view, making copies may be undesirable because as soon as there are two copies, it is unclear which one is the "correct" one if either is changed. Proverb: "A person with one watch always knows what time it is. A person with two watches is never sure." Avoid making copies.

Simple Reference Parameter Example — Swap()
The standard example of reference parameters is a Swap() function which exchanges the values of two `ints`. It's a simple function, but it does need to change the caller's memory which is the key feature of pass by reference.

Swap() Function
The values of interest for Swap() are two `ints`. Therefore, Swap() **does not** take `ints` as its parameters. It takes a pointers to `int` — (`int*`)'s. In the body of Swap() the parameters, a and b, are dereferenced with * to get at the actual (`int`) values of interest.
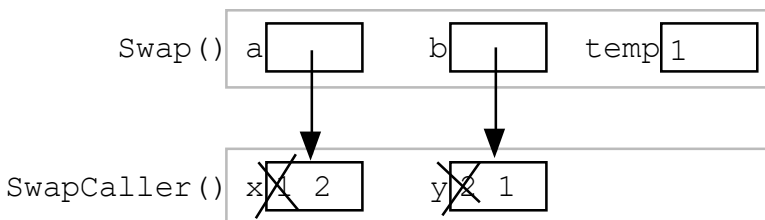
```
void Swap(int* a, int* b) {
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

Swap() Caller
To call Swap(), the caller must pass pointers to the values of interest...

```
void SwapCaller() {
    int x = 1;
    int y = 2;

    Swap(&x, &y);   // Use & to pass pointers to the int values of interest
                    //  (x and y).
}
```


The parameters to Swap() are pointers to values of interest which are back in the caller's locals. The Swap() code can dereference the pointers to get back to the caller's memory to exchange the values. In this case, Swap() follows the pointers to exchange the values in the variables x and y back in SwapCaller(). Swap() will exchange any two ints given pointers to those two ints.

Swap() With Arrays
Just to demonstrate that the value of interest does not need to be a simple variable, here's a call to Swap() to exchange the first and last `ints` in an array. Swap() takes `int*`'s, but the `ints` can be anywhere. An `int` inside an array is still an `int`.

```
void SwapCaller2() {
    int scores[10];
    scores[0] = 1;
    scores[9[ = 2;
    Swap(&(scores[0]), &(scores[9]));// the ints of interest do not need to be
            // simple variables -- they can be any int. The caller is responsible
            // for computing a pointer to the int.
```

The above call to Swap() can be written equivalently as `Swap(scores, scores+9)` due to the array syntax in C. You can ignore this case if it is not familiar to you — it's not an important area of the language and both forms compile to the exact same thing anyway.

Is The & Always Necessary?

When passing by reference, the caller does not always need to use & to compute a new pointer to the value of interest. Sometimes the caller already has a pointer to the value of interest, and so no new pointer computation is required. The pointer to the value of interest can be passed through unchanged.

For example, suppose B() is changed so it calls a C() function which adds 2 to the value of interest...

```
// Takes the value of interest by reference and adds 2.
void C(int* worthRef) {
    *worthRef = *worthRef + 2;
}

// Adds 1 to the value of interest, and calls C().
void B(int* worthRef) {
    *worthRef = *worthRef + 1; // add 1 to value of interest as before

    C(worthRef);            // NOTE no & required. We already have
                            // a pointer to the value of interest, so
                            // it can be passed through directly.
}
```

What About The & Bug TAB?

All this use of & might make you nervous — are we committing the & bug from Section 2? No, it turns out the above uses of & are fine. The & bug happens when an & passes a pointer to local storage from the callee back to its caller. When the callee exits, its local memory is deallocated and so the pointer no longer has a pointee. In the above, correct cases, we use & to pass a pointer from the caller to the callee. The pointer remains valid for the callee to use because the caller locals continue to exist while the callee is running. The pointees will remain valid due to the simple constraint that the caller can only exit sometime after its callee exits. Using & to pass a pointer to local storage from the caller to the callee is fine. The reverse case, from the callee to the caller, is the & bug.

The ** Case

What if the value of interest to be shared and changed between the caller and callee is already a pointer, such as an `int*` or a `struct fraction*`? Does that change the rules for setting up reference parameters? No. In that case, there is no change in the rules. They operate just as before. The reference parameter is still a pointer to the value of interest, even if the value of interest is itself a pointer. Suppose the value of interest is `int*`. This means there is an `int*` value which the caller and callee want to share and change. Then the reference parameter should be an `int**`. For a `struct fraction*` value of interest, the reference parameter is `struct fraction**`. A single dereference (*) operation on the reference parameter yields the value of interest as it did in the simple cases. Double pointer (**) parameters are common in linked list or other pointer manipulating code were the value of interest to share and change is itself a pointer, such as a linked list head pointer.

Reference Parameter Summary
Passing by value (copying) does not allow the callee to communicate back to its caller
and has also has the usual disadvantages of making copies. Pass by reference uses
pointers to avoid copying the value of interest, and allow the callee to communicate back
to the caller.

For pass by reference, there is only one copy of the value of interest, and pointers to that
one copy are passed. So if the value of interest is an int, its reference parameter is an int*.
If the value of interest is a struct fraction*, its reference parameters is a struct fraction**.
Functions use the dereference operator (*) on the reference parameter to see or change the
value of interest.

Section 3 — Extra Optional Material

Extra: Reference Parameters in Java
Because Java has no */& operators, it is not possible to implement reference parameters
in Java directly. Maybe this is ok — in the OOP paradigm, you should change objects by
sending them messages which makes the reference parameter concept unnecessary. The
caller passes the callee a (shallow) reference to the value of interest (object of interest?),
and the callee can send it a message to change it. Since all objects are intrinsically
shallow, any change is communicated back to the caller automatically since the object of
interest was never copied.

Extra: Reference Parameters in C++
Reference parameters are such a common programming task that they have been added as
an official feature to the C++ language. So programming reference parameters in C++ is
simpler than in C. All the programmer needs to do is syntactically indicate that they wish
for a particular parameter to be passed by reference, and the compiler takes care of it. The
syntax is to append a single '&' to right hand side of the parameter type. So an `int`
parameter passes an integer by value, but an `int&` parameter passes an integer value by
reference. The key is that the compiler takes care of it. In the source code, there's no
additional fiddling around with &'s or *'s. So Swap() and SwapCaller() written with C++
look simpler than in C, even though they accomplish the same thing...

```
void Swap(int& a, int& b) {    // The & declares pass by reference
    int temp;

    temp = a;    // No *'s required -- the compiler takes care of it
    a = b;
    b = temp;
}

void SwapCaller() {
    int x = 1;
    int y = 2;

    Swap(x, y); // No &'s required -- the compiler takes care of it
}
```

The types of the various variables and parameters operate simply as they are declared (int in this case). The complicating layer of pointers required to implement the reference parameters is hidden. The compiler takes care of it without allowing the complication to disturb the types in the source code.

Section 4 —
Heap Memory

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

> *Lifetime.* Because the programmer now controls exactly when memory is allocated and deallocated, it is possible to build a data structure in memory, and return that data structure to the caller. This was never possible with local memory which was automatically deallocated when the function exited.

> *Size.* The size of allocated memory can be controlled with more detail. For example, a string buffer can be allocated at run-time which is exactly the right size to hold a particular string. With local memory, the code is more likely to declare a buffer size 1000 and hope for the best. (See the StringCopy() example below.)

The disadvantages of heap memory are...

> *More Work.* Heap allocation needs to arranged explicitly in the code which is just more work.

> *More Bugs.* Because it's now done explicitly in the code, realistically on occasion the allocation will be done incorrectly leading to memory bugs. Local memory is constrained, but at least it's never **wrong**.

Nonetheless, there are many problems that can only be solved with heap memory, so that's that way it has to be. In languages with garbage collectors such as Perl, LISP, or Java, the above disadvantages are mostly eliminated. The garbage collector takes over most of the responsibility for heap management at the cost of a little extra time taken at run-time.
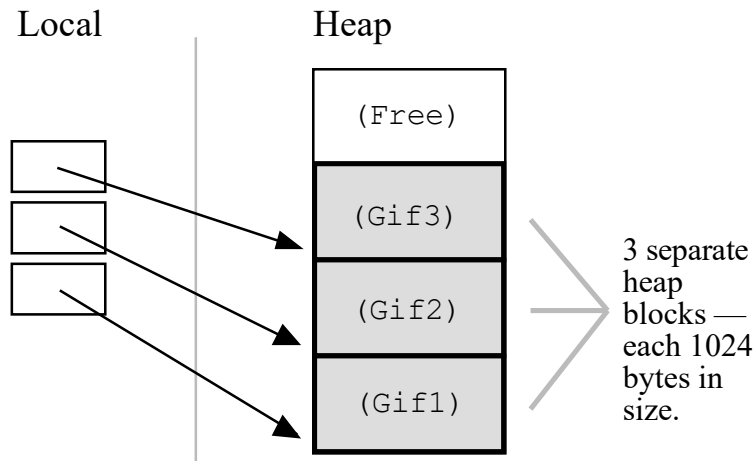
What Does The Heap Look Like?
Before seeing the exact details, let's look at a rough example of allocation and deallocation in the heap...

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

allocate memory to hold three separate GIF images in the heap each of which takes 1024 bytes of memory. After the three allocation requests, memory might look like...

Local   Heap

(Free)

(Gif3)   3 separate
                         heap
(Gif2)   blocks —
                         each 1024
   bytes in
(Gif1)   size.

Each allocation request reserves a contiguous area of the requested size in the heap and returns a pointer to that new block to the program. Since each block is always referred to by a pointer, the block always plays the role of a "pointee" (Section 1) and the program always manipulates its heap blocks through pointers. The heap block pointers are sometimes known as "base address" pointers since by convention they point to the base (lowest address byte) of the block.

In this example, the three blocks have been allocated contiguously starting at the bottom of the heap, and each block is 1024 bytes in size as requested. In reality, the heap manager can allocate the blocks wherever it wants in the heap so long as the blocks do not overlap and they are at least the requested size. At any particular moment, some areas in the heap have been allocated to the program, and so are "in use". Other areas have yet to be committed and so are "free" and are available to satisfy allocation requests. The heap manager has its own, private data structures to record what areas of the heap are committed to what purpose at any moment The heap manager satisfies each allocation request from the pool of free memory and updates its private data structures to record which areas of the heap are in use.

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

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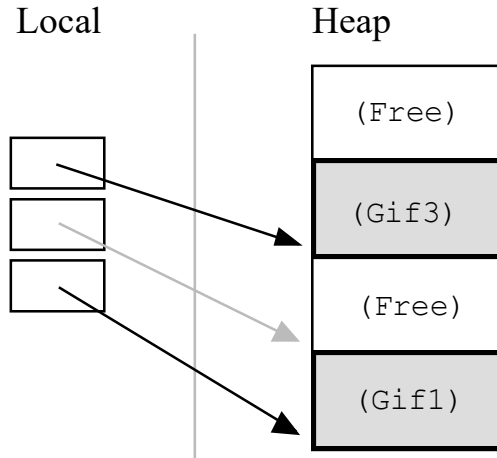
After the deallocation, the pointer continues to point to the now deallocated block. The program must not access the deallocated pointee. This is why the pointer is drawn in gray — the pointer is there, but it must not be used. Sometimes the code will set the pointer to NULL immediately after the deallocation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

Programming The Heap

Programming the heap looks pretty much the same in most languages. The basic features are....

- The heap is an area of memory available to allocate areas ("blocks") of memory for the program.

- There is some "heap manager" library code which manages the heap for the program. The programmer makes requests to the heap manager, which in turn manages the internals of the heap. In C, the heap is managed by the ANSI library functions malloc(), free(), and realloc().

- The heap manager uses its own private data structures to keep track of which blocks in the heap are "free" (available for use) and which blocks are currently in use by the program and how large those blocks are. Initially, all of the heap is free.

- The heap may be of a fixed size (the usual conceptualization), or it may appear to be of a fixed but extremely large size backed by virtual memory. In either case, it is possible for the heap to get "full" if all of its memory has been allocated and so it cannot satisfy an allocation request. The allocation function will communicate this run-time condition in some way to the program — usually by returning a NULL pointer or raising a language specific run-time exception.

- The allocation function requests a block in the heap of a particular size. The heap manager selects an area of memory to use to satisfy the request, marks that area as "in use" in its private data structures, and returns a pointer to the heap block. The caller is now free to use that memory by dereferencing the pointer. The block is guaranteed to be reserved for the sole use of the caller — the heap will not hand out that same area of memory to some other caller. The block does not move around inside the

heap — its location and size are fixed once it is allocated. Generally, when a block is allocated, its contents are random. The new owner is responsible for setting the memory to something meaningful. Sometimes there is variation on the memory allocation function which sets the block to all zeros (calloc() in C).

- The deallocation function is the opposite of the allocation function. The program makes a single deallocation call to return a block of memory to the heap free area for later re-use. Each block should only be deallocated once. The deallocation function takes as its argument a pointer to a heap block previously furnished by the allocation function. The pointer must be exactly the same pointer returned earlier by the allocation function, not just any pointer into the block. After the deallocation, the program must treat the pointer as bad and not access the deallocated pointee.

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

`void* malloc(unsigned long size);` The malloc() function takes an unsigned integer which is the requested size of the block measured in bytes. Malloc() returns a pointer to a new heap block if the allocation is successful, and NULL if the request cannot be satisfied because the heap is full. The C operator sizeof() is a convenient way to compute the size in bytes of a type — `sizeof(int)` for an `int` pointee, `sizeof(struct fraction)` for a `struct fraction` pointee.

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

Simple Heap Example
Here is a simple example which allocates an `int` block in the heap, stores the number 42 in the block, and then deallocates it. This is the simplest possible example of heap block allocation, use, and deallocation. The example shows the state of memory at three different times during the execution of the above code. The stack and heap are shown separately in the drawing — a drawing for code which uses stack and heap memory needs to distinguish between the two areas to be accurate since the rules which govern the two areas are so different. In this case, the lifetime of the local variable intPtr is totally separate from the lifetime of the heap block, and the drawing needs to reflect that difference.

```
void Heap1() {
   int* intPtr;
   // Allocates local pointer local variable (but not its pointee)
   // T1
```

<p style="text-align:center">Local   Heap</p>
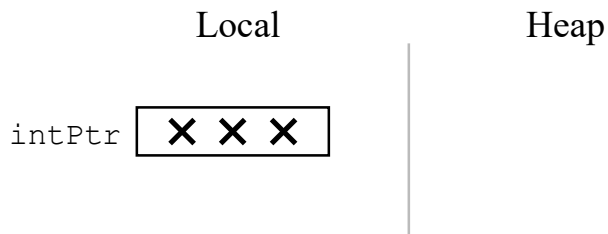
intPtr ☒ ☒ ☒

```
   // Allocates heap block and stores its pointer in local variable.
   // Dereferences the pointer to set the pointee to 42.
   intPtr = malloc(sizeof(int));
   *intPtr = 42;
   // T2
```

<p style="text-align:center">Local   Heap</p>
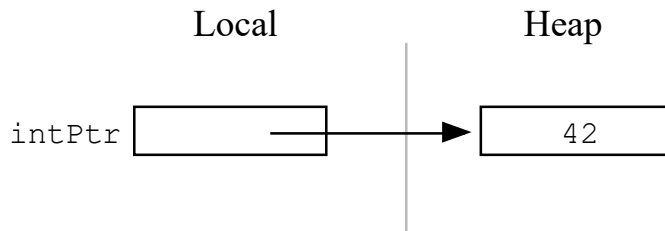
intPtr ⟶ 42

```
   // Deallocates heap block making the pointer bad.
   // The programmer must remember not to use the pointer
   // after the pointee has been deallocated (this is
   // why the pointer is shown in gray).
   free(intPtr);
   // T3
```

<p style="text-align:center">Local   Heap</p>

intPtr ⟶

```
}
```
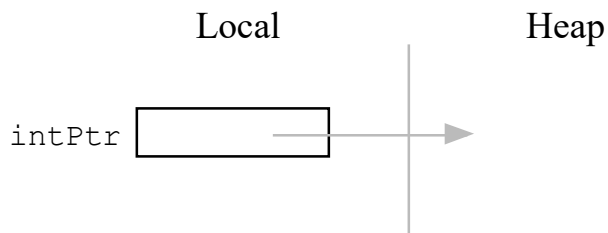
Simple Heap Observations

- After the allocation call allocates the block in the heap. The program stores the pointer to the block in the local variable intPtr. The block is the "pointee" and intPtr is its pointer as shown at T2. In this state, the pointer may be dereferenced safely to manipulate the pointee. The pointer/pointee rules from Section 1 still apply, the only difference is how the pointee is initially allocated.

- At T1 before the call to malloc(), intPtr is uninitialized does not have a pointee — at this point intPtr "bad" in the same sense as discussed in Section 1. As before, dereferencing such an uninitialized pointer is a common, but catastrophic error. Sometimes this error will crash immediately (lucky). Other times it will just slightly corrupt a random data structure (unlucky).

- The call to free() deallocates the pointee as shown at T3. Dereferencing the pointer after the pointee has been deallocated is an error. Unfortunately, this error will almost never be flagged as an immediate run-time error. 99% of the time the dereference will produce reasonable results 1% of the time the dereference will produce slightly wrong results. Ironically, such a rarely appearing bug is the most difficult type to track down.

- When the function exits, its local variable intPtr will be automatically deallocated following the usual rules for local variables (Section 2). So this function has tidy memory behavior — all of the memory it allocates while running (its local variable, its one heap block) is deallocated by the time it exits.

Heap Array

In the C language, it's convenient to allocate an array in the heap, since C can treat any pointer as an array. The size of the array memory block is the size of each element (as computed by the sizeof() operator) multiplied by the number of elements (See CS Education Library/101 The C Language, for a complete discussion of C, and arrays and pointers in particular). So the following code heap allocates an array of 100 `struct fraction`'s in the heap, sets them all to 22/7, and deallocates the heap array...

```
void HeapArray() {
   struct fraction* fracts;
   int i;

   // allocate the array
   fracts = malloc(sizeof(struct fraction) * 100);

   // use it like an array -- in this case set them all to 22/7
   for (i=0; i<99; i++) {
      fracts[i].numerator = 22;
      fracts[i].denominator = 7;
   }

   // Deallocate the whole array
   free(fracts);
}
```

Heap String Example

Here is a more useful heap array example. The StringCopy() function takes a C string, makes a copy of that string in the heap, and returns a pointer to the new string. The caller takes over ownership of the new string and is responsible for freeing it.

```
/*
 Given a C string, return a heap allocated copy of the string.
 Allocate a block in the heap of the appropriate size,
 copies the string into the block, and returns a pointer to the block.
 The caller takes over ownership of the block and is responsible
 for freeing it.
*/
char* StringCopy(const char* string) {
   char* newString;
   int len;

   len = strlen(string) + 1;              // +1 to account for the '\0'
   newString = malloc(sizeof(char)*len);  // elem-size * number-of-elements
   assert(newString != NULL); // simplistic error check (a good habit)
   strcpy(newString, string); // copy the passed in string to the block

   return(newString);    // return a ptr to the block
}
```

Heap String Observations

StringCopy() takes advantage of both of the key features of heap memory...

> *Size.* StringCopy() specifies, at run-time, the exact size of the block needed to store the string in its call to malloc(). Local memory cannot do that since its size is specified at compile-time. The call to sizeof(char) is not really necessary, since the size of char is 1 by definition. In any case, the example demonstrates the correct formula for the size of an array block which is *element-size * number-of-elements.*

> *Lifetime.* StringCopy() allocates the block, but then passes ownership of it to the caller. There is no call to free(), so the block continues to exist even after the function exits. Local memory cannot do that. The caller will need to take care of the deallocation when it is finished with the string.

Memory Leaks

What happens if some memory is heap allocated, but never deallocated? A program which forgets to deallocate a block is said to have a "memory leak" which may or may not be a serious problem. The result will be that the heap gradually fill up as there continue to be allocation requests, but no deallocation requests to return blocks for re-use. For a program which runs, computes something, and exits immediately, memory leaks are not usually a concern. Such a "one shot" program could omit all of its deallocation requests and still mostly work. Memory leaks are more of a problem for a program which runs for an indeterminate amount of time. In that case, the memory leaks can gradually fill the heap until allocation requests cannot be satisfied, and the program stops working or crashes. Many commercial programs have memory leaks, so that when run for long enough, or with large data-sets, they fill their heaps and crash. Often the error detection and avoidance code for the heap-full error condition is not well tested, precisely because the case is rarely encountered with short runs of the program — that's why filling the heap often results in a real crash instead of a polite error message. Most compilers have a

"heap debugging" utility which adds debugging code to a program to track every allocation and deallocation. When an allocation has no matching deallocation, that's a leak, and the heap debugger can help you find them.

Ownership

StringCopy() allocates the heap block, but it does not deallocate it. This is so the caller can use the new string. However, this introduces the problem that somebody does need to remember to deallocate the block, and it is not going to be StringCopy(). That is why the comment for StringCopy() mentions specifically that the caller is taking on **ownership** of the block. Every block of memory has exactly one "owner" who takes responsibility for deallocating it. Other entities can have pointers, but they are just sharing. There's only one owner, and the comment for StringCopy() makes it clear that ownership is being passed from StringCopy() to the caller. Good documentation always remembers to discuss the ownership rules which a function expects to apply to its parameters or return value. Or put the other way, a frequent error in documentation is that it forgets to mention, one way or the other, what the ownership rules are for a parameter or return value. That's one way that memory errors and leaks are created.

Ownership Models

The two common patterns for ownership are...

> *Caller ownership.* The caller owns its own memory. It may pass a pointer to the callee for sharing purposes, but the caller retains ownership. The callee can access things while it runs, and allocate and deallocate its own memory, but it should not disrupt the caller's memory.

> *Callee allocated and returned.* The callee allocates some memory and returns it to the caller. This happens because the result of the callee computation needs new memory to be stored or represented. The new memory is passed to the caller so they can see the result, and the caller must take over ownership of the memory. This is the pattern demonstrated in StringCopy().

Heap Memory Summary

Heap memory provides greater control for the programmer — the blocks of memory can be requested in any size, and they remain allocated until they are deallocated explicitly. Heap memory can be passed back to the caller since it is not deallocated on exit, and it can be used to build linked structures such as linked lists and binary trees. The disadvantage of heap memory is that the program must make explicit allocation and deallocate calls to manage the heap memory. The heap memory does not operate automatically and conveniently the way local memory does.