CMSC 202 | **More about Pointers** | Spring 1996
Sections 101, 102 & 103 | Handout #11

This handout covers a number of miscellaneous pointer–related topics.

1 A Handy Feature

Sometimes it is useful to have your program print out some of the pointer values it is manipulating. The printf function provides an easy way to do this—the %p specifier can be used to print a pointer value in much the same way that the %d specifier can be used to print an integer value. Pointers printed using %p are usually expressed as unsigned hexadecimal numbers. For example, the code:

```
int *ptr;
ptr = malloc(sizeof(int));
printf("The pointer value is %p\n", ptr);
```

might cause the following to be printed:

```
The pointer value is 100000a0
```

You will notice that on our machines a NULL pointer prints as a zero value:

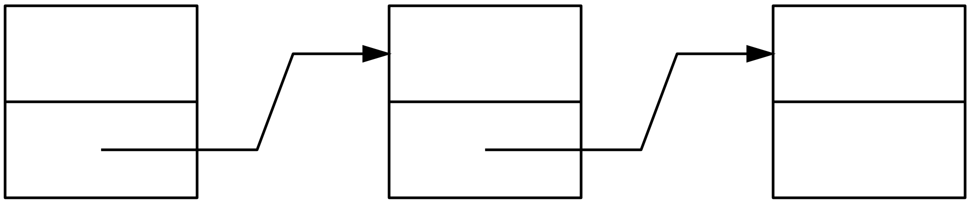
```
int *ptr;
ptr = NULL;
printf("The pointer value is %p\n", ptr);

The pointer value is 0
```

2 A Handy Function

The malloc function is contained in the <stdlib.h> standard library. A second memory allocation function called calloc is also to be found in <stdlib.h>. Calloc takes two arguments—the number of array elements you wish to allocate, and the size of each element. Just like malloc, it returns a pointer to the allocated space of type void *. Unlike malloc, it also sets every word of the allocated space to zero (malloc simply gives a pointer to the space with whatever garbage it used to contain intact). For example, the call `calloc(42, sizeof(int))` allocates enough space to hold an array of forty–two integers, and initializes each of them to zero.

3 Circular Structures

It is often desirable to create structures that contain pointers to other structures of the same type. For example, we might want to create a picture such as:



The intention here is that each of the three `struct`s depicted is of the same type.

When we try to write the `typedef`s for this kind of structure, we find that the `struct` relies on the definition of the pointer type, and the pointer type relies on the definition of the `struct`. Fortunately, C allows us to define a pointer to a `struct` before giving a definition of that `struct`. To do so, we must add a name after the word `struct` (in this case, I've chosen the name `tag`):

```
typedef struct tag *struct_ptr;
```

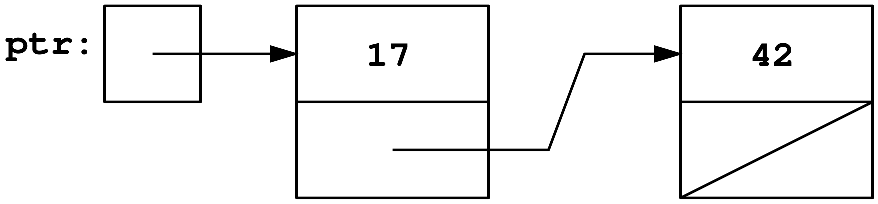
This definition says that an object of type `struct_ptr` may hold a pointer to an object of type `struct tag`. The struct itself can now be defined, again with the inclusion of the name:

```
typedef struct tag {
  int field1;
  struct_ptr field2;
} struct_type;
```

This says that an object of type `struct_type` is a `struct` with two fields: an `int`, and a pointer to another object of type `struct_type`. We can now use these type definitions to build complex interlinked structures:

```
struct_ptr ptr;

ptr = malloc(sizeof(struct_type));
ptr->field1 = 17;
ptr->field2 = malloc(sizeof(struct_type));
ptr->field2->field1 = 42;
ptr->field2->field2 = NULL;
```

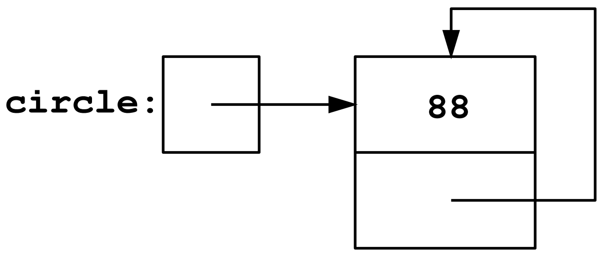
The above code leads to the production of the following picture:



Note that such definitions make it possible to build `struct`s that point to themselves. For example, in combination with the above `typedef`s, the code:

```
struct_ptr circle;

circle = malloc(sizeof(struct_type));
circle->field1 = 88;
circle->field2 = circle;
```

builds the picture:



4 A Memory Model

4.1 Types of Memory

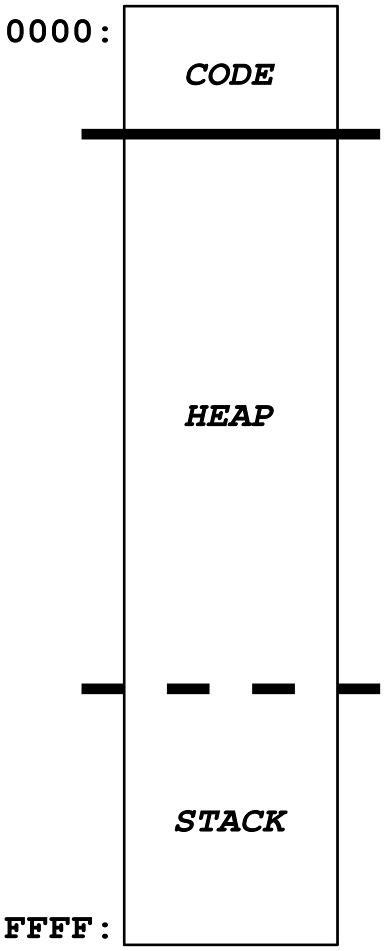


Figure 1: The memory model

When you run your program, the memory it has available to it is divided up into small chunks called **words**. Each word has an **address**, which is simply a number associated with it. These numbers typically start at zero, and run up to some big number. Figure 1 shows a picture of how your program's memory might be organized.[1] In this example, memory is numbered from `0000` at the top to `FFFF` (the hexadecimal representation of 65535, or $2^{16} - 1$) at the bottom.

The C compiler translates your C code into machine code. When you run your program, this machine code is loaded into a portion of memory called the **code segment**. Typically, this area of memory is write–protected, so you can't accidentally overwrite your code. In our example, the code segment is shown at the top of Figure 1.

When your program runs, it needs more space than merely that allocated for the code segment. For example, it needs space for local variables, for handing parameters to sub-

[1] Each operating system and compiler organizes the memory available to your program slightly differently, but the basic ideas presented in this example are generally applicable.

programs, for returning function values, *etc.* A second area of memory, called the **stack**, is used for this purpose. In our example, the stack resides at the bottom of memory. The stack can grow and shrink as the demands of your program change. For example, whenever a function is called, the stack grows to make room for the function's local variables. When the function terminates, the stack shrinks, leaving behind as garbage the space that had been allocated for local variables. When another function call is made, this same space can then be reused for the new function's local variables.

When you call `malloc`, one of `malloc`'s tasks is to allocate a block of space somewhere. The code segment is reserved for code, and for other items that never change. The stack grows and shrinks; you wouldn't want to allocate space on the stack and then have it go away while your were using it. The **heap** is the middle area of memory in Figure 1; it is used for dynamic allocation of memory.

4.2 Pointers and the Memory Model

In this memory model, *a pointer value is a pairing of an address and a type.* The computer can't actually store the base of an arrow in a variable. Rather, it stores the address of the word(s) that have been allocated for the object referenced. Dereferencing a pointer then corresponds (in most cases[2]) to treating the pointer value as an address, and going to that address.

Consider the code:

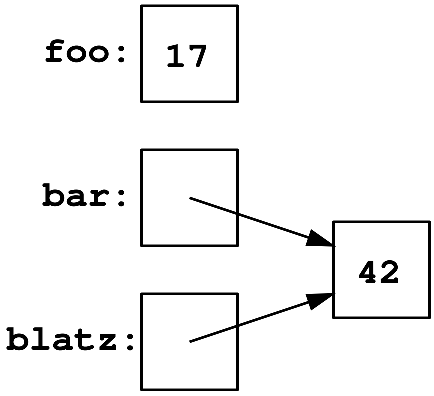
```
int foo;
int_ptr bar;
int_ptr blatz;

foo = 17;
bar = malloc(sizeof(int));
*bar = 42;
blatz = bar;
```

Here is our pictorial representation of the resulting structures:

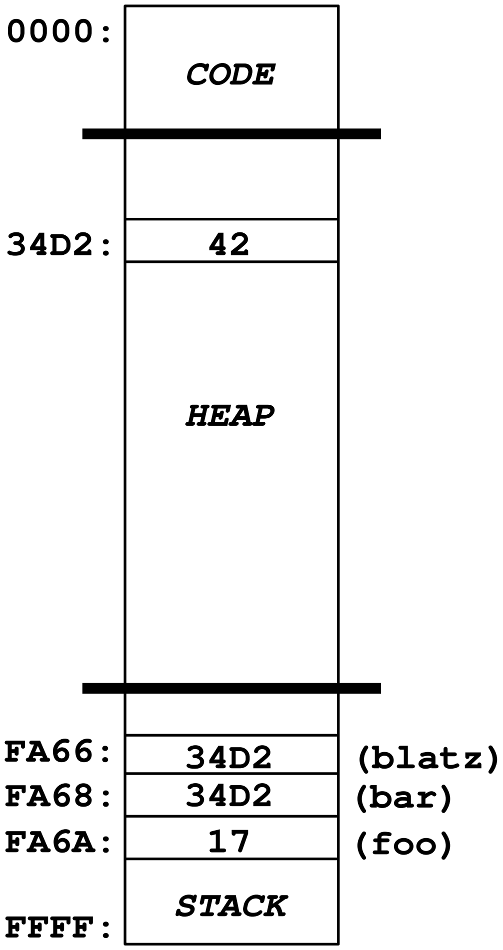
[2]The following code provides a counterexample to this claim:

```
typedef int array[17];
typedef array *arrayptr;
arrayptr blatz;
blatz = malloc(sizeof(array));
(*blatz)[0] = 42;
printf("%p", blatz);
printf("%p", *blatz);
```

The two `printf`s in this code print exactly the same value! This code fragment demonstrates the importance of viewing a pointer not simply as an address, but rather as the combination of an address and a type. Can you explain its output?



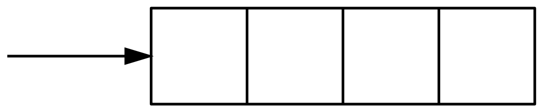
Each of the boxes in the picture represents one or more words of memory at distinct memory locations. The named boxes are allocated in the stack, while the unnamed box is allocated in the heap. Here's how memory might appear once this code has executed:



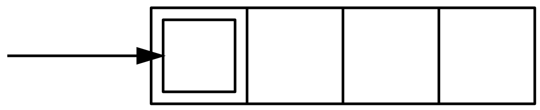
Note that the variable names and addresses are not actually stored in memory; they are shown here merely as points of reference. The diagram shows that each of the three variables has been allocated space on the stack. The space for the integer variable `foo` contains the integer assigned to that variable (17). The pointer variable `bar` contains the address of a memory location in the heap. This latter location contains 42, the value assigned to `*bar`. When `bar` is assigned to `blatz`, the content of the memory location allocated for `bar` is simply copied to the memory location allocated for `blatz`. Because the address of the memory location that contains 42 is copied, the variable `blatz` now refers to the same object.

5 Pointers to Array Elements

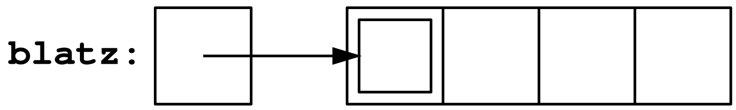
In common C usage, it is rare to see pointers to entire arrays:



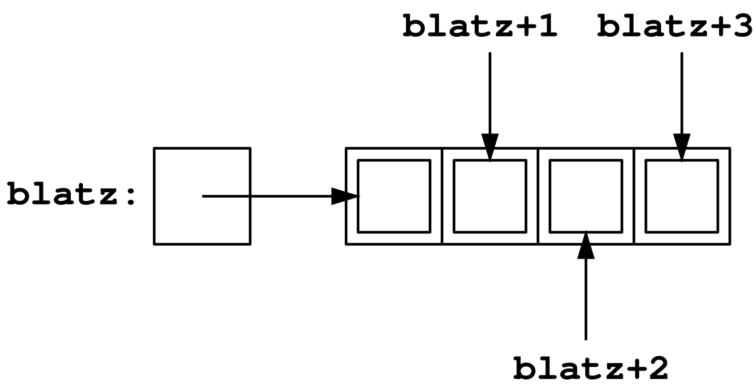
It is much more common to see a dynamically allocated array accessed through a pointer to its zeroth element:



Even though this pointer refers only to a single array element, the use of pointer arithmetic allows each of the other elements to be accessed. For example, if the variable `blatz` contains a pointer to the zeroth element of an array:



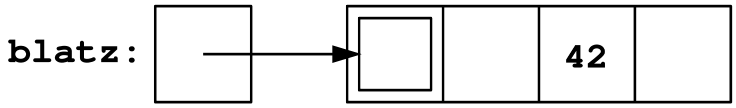
Then the expression `blatz + 1` denotes a pointer to the `[1]` element of the array, and so forth:



Furthermore, we can use the equivalence:

$$var\texttt{[}index\texttt{]} \quad \equiv \quad \texttt{*(}var \texttt{ + } index\texttt{)}$$

to access these elements using subscript notation. For example, the statement `blatz[2] = 42;` is equivalent to the statement `*(blatz + 2) = 42;` and it will therefore produce the picture:



Does this mean that `blatz` is an array, or that `blatz` is somehow equivalent to an array? Not at all. It simply means that the same subscript notation can be used to access both types of structure.

6 Using `malloc`

In C, the type `void *` is used to refer to an object that you know to be a pointer, but where the type of the object referenced by that pointer is unknown. If you examine the prototype for `malloc` in `<stdlib.h>`, you will see that its return type is defined to be `void *`. That means that the compiler has no type information about the block of space referenced by the return value. For this reason, it is possible to create two very different kinds of pointer out of the same kind of call to `malloc`.

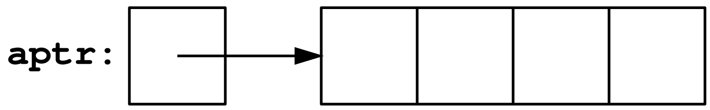
For example, given the following definition and declarations:

```
typedef int array[4];

array *aptr;
int *iptr;
```

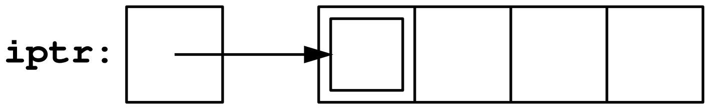
each of the following assignments is legal:

```
aptr = malloc(sizeof(array));
iptr = malloc(sizeof(array));
```

The first statement builds the picture:



while the second builds the picture:



How is it possible for both of these to be legal? Recall that in the memory model, a pointer is the combination of an address and a type. But by declaring itself as a `void *` function, `malloc` throws away all type information about its return value; the only information that remains is the address. And since the start of an array and that array's zeroth element necessarily reside at the same memory location, they must have the same address.

In these examples, the type information is added back in only when the left–hand side of the assignment statement is taken into consideration. Since the compiler knows the respective types of `aptr` and `iptr`, it is able to handle the result of the assignment correctly in each case.

Some people prefer to make the return type of each individual call to `malloc` explicit, rather than leaving the return type to be derived implicitly from the type of the left–hand side of the assignment statement. This can be accomplished through coercion.[3] Using this technique with our example leads to the following code:

```
aptr = (array *) malloc(sizeof(array));
iptr = (int *) malloc(sizeof(array));
```

Making these types explicit can help you to think about your code. In old–style C, such coercions were important. In ANSI C, they are optional. Which approach you decide to use is a matter of personal style.

7 Dangers of the & Operator

The `&` operator may be applied to any object for which memory has been allocated. That means that `&` can return a pointer to an object that later goes away. There are two ways that such an event can occur:

1. Space on the system stack is deallocated when the function in which it was allocated terminates.
2. Space in the heap is deallocated by a call to `free`.

[3]An object of one type can be coerced to a new type by placing the name of the new type in front of the object and in parentheses.

Consider for example the following function:

```
int *
badfn(void)
{
  int local;

  return(&local);
}
```

`Badfn` is an example of a function that returns a reference to a local variable, which is then deallocated when the function exits. Thus, the caller gets back an address on the stack that is no longer (necessarily) being used to hold an `int`. Any attempt to dereference such a pointer is likely to wreak havoc, because there is no way to know to what purpose the space referenced is being put. It is your responsibility as a programmer to avoid creating and using pointers to space that has been deallocated; the compiler is unable to detect such situations for you.

8 Pass–by–value and Pass–by–reference

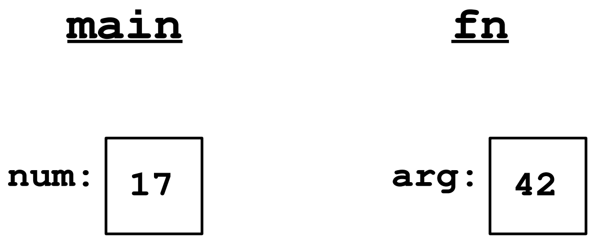
One of the decisions that the designers of a programming language must make is what it means to pass an argument to a function. For example, the designers might decide that if while executing a function you change the value of one of the function's parameters, that change will also affect the argument that was handed to the function. Parameters that behave in this way are said to be *pass–by–reference* parameters. [If you have ever programmed in Pascal, you will recognize `var` parameters as pass–by–reference parameters.] If on the other hand a change to one of a function's formal parameters causes no change in the argument handed to the function, the parameter is said to be *pass–by–value*.

In C, **all arguments are pass–by–value**. Consider for example the following code:

```
void
fn(int arg)
{
  printf("At the top of fn, arg = %d\n", arg);
  arg = 42;
  printf("At the bottom of fn, arg = %d\n", arg);
}
```

```
main()
{
  int num;

  num = 17;
  printf("At the top of main, num = %d\n", num);
  fn(num);
  printf("At the bottom of main, num = %d\n", num);
}
```

Pictorially, this code creates:

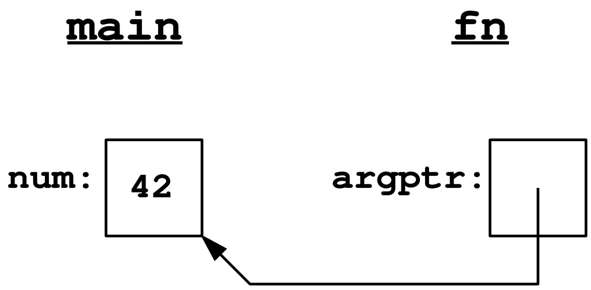


When this code runs, the value of `arg` is 17 at the top of `fn`, but 42 at the bottom of `fn`. However, in `main` the variable `num` has value 17 throughout; it is unaffected by the assignment of 42 to `arg` in `fn`. Another way to think of this is to view all function parameters in C as local variables that are declared inside the function.

Because C does not provide pass–by–reference parameters, sometimes programmers will use pointers to simulate pass–by–reference. To do this, it is necessary to pass the address of the desired variable as the argument to the function. Inside the function, the variable is accessed (and possibly modified) by dereferencing the pointer. Here is an example:

```
void
fn(int *argptr)
{
  printf("At the top of fn, *argptr = %d\n", *argptr);
  *argptr = 42;
  printf("At the bottom of fn, *argptr = %d\n", *argptr);
}

main()
{
  int num;

  num = 17;
  printf("At the top of main, num = %d\n", num);
  fn(&num);
  printf("At the bottom of main, num = %d\n", num);
}
```

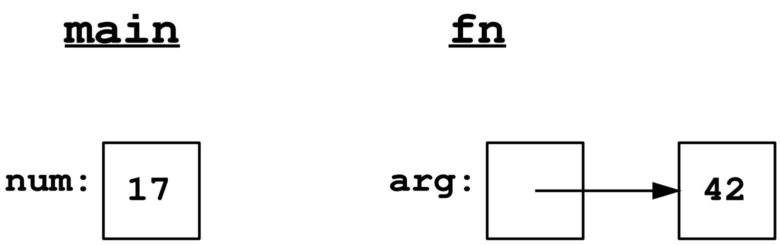
This code creates the following picture:



When this code runs, the change to `*argptr` causes a change to `num`, because `argptr` refers to `num`. Note that if `argptr` itself is changed, num cannot then be affected, even though `argptr` is a pointer. For example, the code:

```
void
fn(int *argptr)
{
  printf("At the top of fn, *argptr = %d\n", *argptr);
  argptr = malloc(sizeof(int));
  *argptr = 42;
  printf("At the bottom of fn, *argptr = %d\n", *argptr);
}

main()
{
  int num;

  num = 17;
  printf("At the top of main, num = %d\n", num);
  fn(&num);
  printf("At the bottom of main, num = %d\n", num);
}
```

creates the situation:



The value of `num` in `main` is unchanged in this example. The difference between this example and the previous one is that there only the object referred to by `argptr` was modified, while here `argptr` itself is modified.