MODULE 5 : USING POINTERS

Professor : Dave Houtman
Office: T323
Office Hrs: Monday 15:30 – 16:00
Wednesday 15:30 – 16:00
Friday: 15:30 – 16:00
Email: houtmad@algonquincollege.com

4.1 Review of primitive and reference data types

- When you declare a ***primitive*** data type in any language, like

```
int counter;
```

you are doing a number of things, of which the three most important are:

1. You are *assigning a name to a place in memory*;
2. You are *determining the amount of memory that will be allocated*, and therefore the range of values that can be stored in that location, according to the data type;
3. You are making a *statement of intent* about how you intend to use this variable, which the compiler/interpreter will then use to verify the correctness of any code that accesses this location in memory.

There are often other issues at play, such as the scope of the variable or its access modifier/storage class, or whether it is automatically initialize to 0 or not. But the above three at the main ones that concern us.

4.1 Review of primitive and reference data types

- When you declare a ***reference*** data type such as a pointer, you're doing much the same thing

```
int *xyz;
```

where the asterisk indicates that `xyz` is a pointer. Once again,

1. You are *assigning a name* (in this case called `xyz`) *to a place in memory*.
2. You are *determining the amount of memory that will be allocated*, which in this case is dependent upon the size of an address. This will be either 32-bits or 64-bits depending on your PC's bus-width;
3. You are making a *statement of intent* about how you intend to use this variable, which the compiler/interpreter will then use to verify the correctness of any code that accesses this location in memory.

As with primitive data types, there are other issues at play. But this list outlines the fundamental features of a pointer.

4.1 Review of primitive and reference data types

- When we declare a primitive data type, we can graphically show the declaration as follows:

```
char ch;
```

means

Name: **`ch`**

Address: **`0x5F237FF0`**
(which is generally unknown, and not *usually* of any concern)

`0x00000000`

Value: (which could be garbage depending on whether the compiler initializes the variable or not)

4.1 Review of primitive and reference data types

- When we declare a pointer, we can similarly represent the situation graphically as follows:

```
char *pCh;
```

means

Name: **`*pCh`**

Address: **`0x5F237FF0`** (which is again unknown, and still not of any real concern)

`0x00000000`

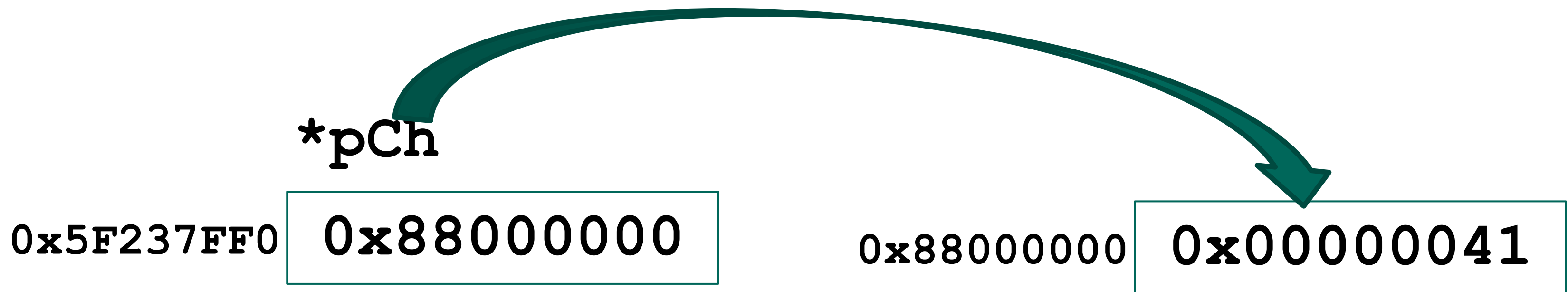
Value: (which could again initially be set to garbage)

- In C and C++, the asterisk (*) is used to signal a pointer declaration. Take the asterisk as meaning "*is a pointer to a*". Read from right to left, the declaration at top says that "`pCh` *is a pointer to a* `char`".
- * is referred to as the ***dereferencing operator***.

4.2 What is a pointer?

If this was all there was to pointers, they would not be terribly useful. What use is it to be able to directly manipulate an address, if that's all that happens? The power of pointers comes from the fact that they allow the programmer to access a CPU's ***indirect addressing mode***.

In indirect addressing mode the CPU 'knows' that operations on a pointer variable are in fact directed not at the address stored ***in*** *that variable, but at the information being pointed to* ***by*** *that variable.*



4.2 What is a pointer?

- So in a very real sense, a pointer is just a place in memory, with a name, one that contains an address. However, when used with the CPU's indirect addressing mode, a pointer can be used to indirectly manipulate a location in memory by 'pointing to it' in the pointer itself. There are three potential complications to this simple definition:

1) What is being pointed to must have a type, otherwise the compiler cannot know how many bytes to read, or how to interpret the bytes being pointed to. When you declare something like:

```
char *pCh;
```

   you are informing the compiler that the pointer points to a single byte, the contents of which are to be interpreted as a `char` data type;

2) The memory *being pointed to* must be reserved if it is to be manipulated indirectly. In the above declaration, *only the pointer itself has memory reserved by the declaration itself; what it is pointing to does not*;

3) In actual use, the pointer conceptually becomes, not just a variable containing an address, but *the thing the pointer is pointing to itself*.

4.2 What is a pointer?

- Because it has the special job of *pointing to* data (unlike primitive data types, which just *are* the data) a pointer can be thought of as being two things as once:

  1. It contains an address…

  So a pointer is just a number like (on a 32-bit processor) `0x82FE8000`. In this sense, it works no differently than any other declaration, like:

```
unsigned long foo = 0x82FE8000;  //unsigned long type
```

  except that now, instead of the memory location storing a long data type, the data stored in the pointer's memory is an address instead.

4.2 What is a pointer?

2. ...but that address points to a particular data type

- In its actual usage, we're rarely concerned with the address stored *in* a pointer; its what we're pointing *to* that really matters. That's why we must specify that our pointer is pointing to a `char`, `int`, `float`, … *The compiler needs to know the data type we're pointing to*, since it needs to know things like the type, range, storage class of that object, etc.

- So in a sense, pointers are 'split personality' data types, having two different natures at once. When using pointers, we need to keep both aspects of this indirect data type in mind .

4.2 What is a pointer?

- This 'dual nature' is reflected in the following two slightly different ways you can write a pointer declaration. (Note that both declarations are syntactically correct, since the location of the * and space inside the declaration doesn't matter)

1. ***Interpretation 1***:

```
char *myString
```

`myString` *is* a pointer, as the asterisk indicates. Therefore it literally *is* an address, in the same way that we think of the declaration `int myInt` as indicating that `myInt` *is* an integer.

2. ***Interpretation 2***:

```
char* myString
```

`myString` *points to* a character data type. So `myString` essentially *talks* to that character, *not* just to the address stored inside `myString`.

4.3 Pointer declarations

- Because everything stored in your computer has an address, virtually everything can have a pointer set to it. So the list of potential pointer declarations is virtually endless. There are

  - pointers to primitive data types
  - pointers using `typedef`ed values
  - pointers to constant values, and constant pointers to values
  - pointers to reference data types i.e. pointer to pointers
  - pointers to strings
  - pointers to arrays
  - pointers to arrays of pointers
  - pointers to functions
  - pointers to functions that return an array of pointers to other functions
  - pointers to user-defined types, called structures
  - pointers to fixed places in memory, such as the real time clock (RTC)
  - pointers to allocated blocks of memory on the heap
  - pointers to memory in the stack
  - *etc.*

4.3 Pointer declarations – Segmentation Faults

- Pointers may be declared without initialization; the following is perfectly legitimate:

```
int *pInt;
```

However, any attempt to use `*pInt` at this point will most likely result in a ***segmentation fault***, which is a special type of error that results when a program attempts to access memory for which it does not have permission.

Prior to actual use, a pointer *must* be assigned a legitimate address value: it must point to some data object that has been assigned space by the OS. Alternately, certain C functions allow the user to reserve space on the heap, and the address of the start of this space can be stored in `pInt`. Note that the above declaration may result in `pInt` being set to zero by default, depending on the storage class of the declaration. This too will result in a segmentation fault, since 0x00000000 is not 'safe' memory to talk to.

4.3 Pointer declarations—assigning an address

- In order to initialize a pointer, you require two things: the name of the data being pointed to, and its address (to initialize the pointer itself)
- As already discussed, the **&** operator is used to return *the address* of a variable.

```
int myInt = 1;          // create an int and set it to 1
int* pInt;              // declare a pointer to an int
pInt = &myInt;          // put the address of myInt into
                        // pInt
```

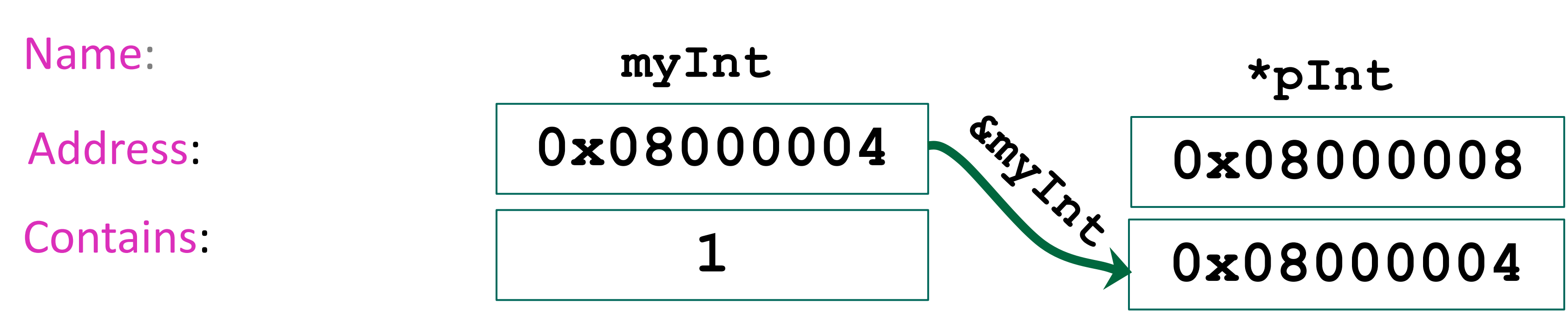
As with most initializations, it is possible to make the declaration and assignment in one line:

```
int myInt = 1;          // value stored in myInt set to 1
int* pInt = &myInt; // declare & assign pInt; *pInt = 1
```

4.3 Pointer declarations—assigning an address

- Note that the type of the pointer must agree with the type of data to which it points

```
int myInt = 1;              // data type is an int
int* pInt = &myInt;         // pointer to an int
```



- Note that the address of a variable is assigned by the OS itself and is generally not of any immediate concern to us. (Values used in all examples are, of course, entirely fictitious.)

4.3 Pointer declarations – assigning a address

- While we're generally not interested in the address of either the pointer itself *or* the address stored *in* it, it is desirable in some situations to be able to set a pointer equal to a fixed value (for example in embedded systems, when you know exactly where things are located and need to talk to memory directly). For example

```
p = 0;      // set address to lowest location in memory
p = NULL;   // same as before, equal to address 0x0
p = (int *) 0xFFFFFFF0;  // p is set to address
           // FFFFFFF0h,  which has been cast
           // to point to an int
```

With the exception of an embedded system, it is generally not advisable to attempt to 'talk' directly to a place in memory whose address is fixed. In the PC world, in any system running under Windows or Linux/Unix, this will certainly generate a segmentation fault.

4.3 Pointer declarations—assigning an address

- There is an important distinction to be made between the following two assignments; make sure you fully understand them. Assume

```
char ch = 'A'.

char *pCh = &ch; // p is a pointer to a char;
                 // p's value is the address of ch
                 // This is correct usage

char *pCh = ch;  // p is a pointer to a char; but
                 // pCh's value is now set to the
                 // value stored in ch.  This is
                 // probably an error, since ch
                 // stores a value like 'A'
                 // (0x00000041), which is not an
                 // accessible address in memory
```

4.3 Pointer declarations—assigning an address

- Note that the declaration of a value and the assignment of its address to a pointer can be made in a single declaration/assignment

```
char ch ='A', *pCh = &ch;
```

This corresponds to the declarations made in the example above: `ch` is declared as a `char` and set to '`A`'; `pCh` is declared as a pointer to a `char`, and set equal to the address of `ch`. The order of the declarations *is* important, since `ch` must exist before we can get the address of it.

4.4 Pointers in action

- As with primitive data types, to use a pointer, simply reference it by name, but without the type declaration. (You must, of course, continue to use the '*' to use the pointer as a pointer; otherwise, you're talking directly to the contents inside the pointer variable, which is presumably just an address). For example, assume

```
int X = 2, *pX = &X;
```

Then

```
*pX = 1;
```

Says: set the value pointed to by `pX` equal to '1'. Since `pX` contains the address of `X`, `X` is now equal to 1.

4.4 Pointers in action

Looking at this in detail, we initially set `*pX` equal to the address of `X`

Name:	X		*pX
Address:	&X		some_addr
Contains:	2	&X →	&X

When we say '`*pX = 1`', we are saying: "Set the value that `pX` points to to the value '1'."

Name:	X		*pX
Address:	&X		some_addr
Contains:	1	←	&X

4.4 Pointers in action – Note

One subtle feature about *the for*m of pointer declarations can make their use especially confusing. This is with regard to the slightly different ways in which the '`*p`' expression gets used.

Consider the following:

In a *primitive* data type declaration like:

```
int myInt;        // declare int
```

we can assign a value to `myInt` using:

```
myInt = 103;     // assign an integer value
```

We can shorten this into a single statement using:

```
int myInt = 103;// declare and assign in one line
```

4.4 Pointers in action – Note

We have exactly the same notation with pointers. We can make a pointer declaration with the line

```
int *pInt;    // declare pInt
```

and then assign the address, like this

```
pInt = &myInt; // assign an address
```

Again, there is a shortcut version of a declaration and an assignment 'all in one go'

```
int *pInt = &myInt;  //declare and assign
```

ALGONQUIN COLLEGE

4.4 Pointers in action – Note

But if you compare the two situations, you'll detect a slight inconsistency in the way the two notations are used.

You set the `int` variable like this:

```
myInt = 103;
```

And use it like this:

```
myInt++;
```

With the pointer, while you use the pointer like this:

```
*pInt = 100;
```

you *don't* set the pointer like this

```
*pInt = &myInt;
```

4.4 Pointers in action – Note

So here's the potential point of confusion. In the shortcut declaration/assignment:

```
int *pInt = &myInt;
```

`*pInt` is being used to *initialize the pointer to the address being pointed to*. But, *once the pointer is declared*, in a statement like:

```
*pInt = 3;
```

`*pInt` talks to the value *being pointed to* at the address already assigned. So the initial declaration is a special and potentially confusing case, since it its usage is different from regular usage after the declaration. The first use stores an address into `pInt`; the second uses that address to change something being pointed to. `*pInt` appears in both cases, but in very different ways.

4.4 Pointers in action – Note

It may make more intuitive sense to write the declaration and assignment as two separate statements:

```
int *pInt;      // pInt is a pointer to an int
pInt = &myInt; // Set pInt (not *pInt) equal to
               // the address of myInt
```

The following, while correct, can be confusing for novices, since it implies that the address is being assigned to `*pInt`, rather than `pInt` itself:

```
int *pInt = &myInt;
```

4.4 Pointers in action – ****Note****

As a rule, always assume the following:

Whenever you see something in the form of a declaration plus assignment, i.e. with the data type being pointed to specified, such as in

```
float *pFloat = …
```

then what appears on the right hand side of the equals sign ***must** be an address*. However, when you see

```
*pFloat  =
```

by itself, without the data type in front, then treat this as equivalent to the value *being pointed to*.

4.4 Pointers in action

- Assume

```
int X = 2, Y = 5, *pX = &X, *pY = &X;
```

Then

```
*pX = 1;      // so X = 1 now
              // also, pY now points to a 1
pY = &Y;      // pY now points to 5
pX = pY;      // pX now contains the address of Y
*pY = X;      // The value pointed to by pY,
              // which contains the address of Y,
              // now equals 1. Hence Y = X (= 1)
```

4.4 Pointers in action

- Assume

```
int X = 2, Y = 5, *pX = &X, *pY = &Y;
```

Then

```
*pX += 1;   // so X = 3 now
*pY *= *pX; // The value pointed to by pY is
            // equal to itself times the value
            // pointed to by pY, ie. 5 * 3 = 15
*pX <<= 3;  // left-shift the value pointed to
            // by pX 3 times, ie. X = 2³*3 = 24
```

- As with all expressions involving shortform operations, the location of the spacing is important. Again, parentheses should be used to disambiguate the programmer's intention.

4.4 Pointers in action

Notice the difference between the following two statements

```
(*p)++;
```

Says: increment the value pointed to by `p` by one. Note that the parenthesis ensure that the dereference operator takes precedence over the post-increment operator. If the parenthesis are different, the statement:

```
*(++p) = 3;
```

says: increment the address value stored in the pointer `p`—and so you're now pointing to a new place in memory—and set the memory it points to equal to 3. This will probably generate a segmentation fault.

(What does `*p++` do? For a useful discussion on the difference between precedence vs. order of evaluation, see:

http://www.eskimo.com/~scs/readings/precvsooe.20010512.html)

4.4 Pointers in action

The expression

```
*p = (float)*q;
```

Says: cast the value pointed to by `q` and set the value pointed to by `p` to it. Of course, if the data types on both sides of the '=' aren't the same, you'll get an error. So the value pointed to by `p` must also be a float.

4.5 Constant pointers and constant values

- Because pointers allow you to talk directly to memory, you can often override the normal limitations imposed on your data by the compiler. This applies to values which were declared constants. The following code flags a warning in `gcc`, but otherwise compiles and executes normally:

```
const int MoL= 42;          // set a constant value
printf("Value of constant int is %d\n", MoL); // 42

int *pMol = &MoL;           // point to the constant value
(*pMol)= 0;                 // change the value pointed to
printf("Value of constant int is now %d\n", MoL); // 0
```

ALGONQUIN COLLEGE

4.5 Constant pointers and constant values

- Note that we can declare the pointer as `const` as well:

```
int MoL;
const int *pMoL = &MoL;
```

  Now the pointer is constant, but the value it points to can change. So `pMol` is fixed, but `*pMol` can be altered.

- To ensure that a pointer to a constant value cannot affect that value, both the pointer and the value it points to must be fixed as `const`:

```
const int pMoL= 42;
const int *const pMoL = &pMoL;
```

  The last statement says (reading from right to left) that `pMoL` is a constant pointer to an integer constant.

Questions

1. If `p` is a pointer then:

`*p`	is what is being pointed to
`&(*p)`	is the address of what is being pointed to
`== p`	which stores the address of what is being pointed to

so `&(*p) == p;`

Similarly, show that `*(&p) == p;` and therefore, in a sense, `&` is the inverse operation of `*`. `&p` takes us 'up' a level; `*p` takes us 'down' one level. And so `p == *&p == &*p;`

ALGONQUIN COLLEGE

4.6 Passing a pointer to a function

- To pass a pointer to a function simply means to pass an address into a function argument declared as a pointer. A simple example helps clarify this. The following function reads in the address of an alphabetic `char` and increments its value by 1.

```
void nextChar (char *pChar){
    if (*pChar == 'z')
        *pChar = 'a';
    else if (*pChar == 'Z')
        *pChar = 'A';
    else
        (*pChar)++;        // incr char value by 1
}
```

4.6 Passing a pointer to a function

- So what gets passed to the function must be the address of a `char`

```
char c = 'a';
printf("Value of c is %c", c);     // 'a'

nextChar (&c);             // pass the address
printf("Value of c is %c", c);     // 'b'
```

- Note that the function prototype for `nextChar()` is

```
void nextChar (char *);
```

but the actual call takes the form

`nextChar (&c);` AND NOT `nextChar(*c);`

*c is what is being pointed to; it is NOT an address

4.6 Passing a pointer to a function

- A string is just an array of characters. When you write:

```
char ch[] = "ABC";
```

You're doing three things at once:

1. Reserving *four* bytes in the memory and setting the first to 'A', and the remaining three locations to 'B', 'C', '0'.
2. Assigning the name `ch` to a place in memory;
3. Setting the value of `ch` equal to the address of the first character, 'A';

So `ch` *points to* an array of 4 chars. To print out the contents of an array, you could use:

To print out this information, use:

```
for (i = 0; i < 10; ++i)
    printf("%c", ch[i]);
```

ALGONQUIN COLLEGE

4.6 Passing a pointer to a function

- This declaration could equally well be written

```
char *ch = "ABC";
```

which says:

1. Reserve *four* bytes in the memory and set the first to 'A', and the remaining three locations to 'B', 'C', '0'.
2. Assigning the name `ch` to a pointer to a char;
3. Setting the value of `ch` equal to the address of the first character, 'A';

To print out the contents of this array using pointers, we could use:

```
for (ch[0]; *ch !='\0'; ch++)
    printf("%c", *ch);
}
```

ALGONQUIN COLLEGE

4.6 Passing a pointer to a function

- In both of the previous examples, `ch` is a variable that stores an address. Now consider the situation in which we wish to pass a string to a function. As explained earlier, a string literal is passed to a function as the address of the first character of the string. So you never actually pass a string to a function: you only pass a copy of the address of the first character of the string to the function. For example:

```
void famousFilmQuotes(char *str){
   printf("%s", str);    // print out the string
}

char *myString = "Soylent Green is people!";
...
famousFilmQuotes(myString);
...
```

4.6 Passing a pointer to a function

- It's worth noting that passing an array to a function is no different than passing a pointer to a function. For example, the `famousFilmQuotes()` example could just as well be written:

```
void famousFilmQuotes(char str[]){
   printf("%s", str);   // print out the string
}

char myString[] = "Soylent Green is people!";
...
famousFilmQuotes(myString);
...
```

4.6 Passing a pointer to a function

- In both cases, it's important to understand *exactly* what's going on here, since this only gets more confusing later. The line:

```
char *myString = "Soylent Green is people!";
```

whether written as a pointer or an array, puts the address of the first character ("S") into `myString`; `myString` now stores an address that points to a `char` (which happens to be the "S".)

The function header

```
void famousFilmQuotes(char *str){...}
```

says that `str` is a pointer to a `char`; so `str` *must* store an address to a `char` work properly.

4.6 Passing a pointer to a function

Finally,

```
famousFilmQuotes(myString);
```

passes the contents of `myString`, the address of the "S", into our function, via **`str`** in

```
void famousFilmQuotes(char *str){
   printf("%s", str);    // print out the string
}
```

So `str` contains the address of the start of the string, and this, in fact, is what `printf` uses to perform its job, printing characters from the start until it finds a `'\0'`.

ALGONQUIN COLLEGE

4.6 Passing a pointer to a function

- This example recalls our the earlier discussions about using `scanf()`. The `scanf()` declaration looks like this:

```
int scanf(const char *format, …);
```

The first argument of `scanf()` is a pointer to a formatting string (like `"%d"`). As with any string, `"%d"` returns the address of the string itself—exactly what the first argument of `scanf()` expects. So "passing the argument"—in this case, a pointer argument to a `const char`—means essentially making the following assignment:

```
const char *format = [starting address of]"%d"
```

Conveniently, C regards any reference to a string literal as equivalent to its address. Hence when you use `scanf("%d",…)` you are setting the contents of the `format` argument equal to the address of the `%` in `"%d"`. The same thing happens with the first argument of `printf()`.

ALGONQUIN COLLEGE

4.6 Passing a pointer to a function

- Let's look at the above example again, this time using some fictitious addresses. Assume the string "`Soylent Green is people`" is located at address `0x7DFE0204`—that is to say, the "S" is stored at this address Therefore, setting

```
char myString[] = "Soylent Green is people!";
```

sets the value stored in `myString` to `7DFE0204`

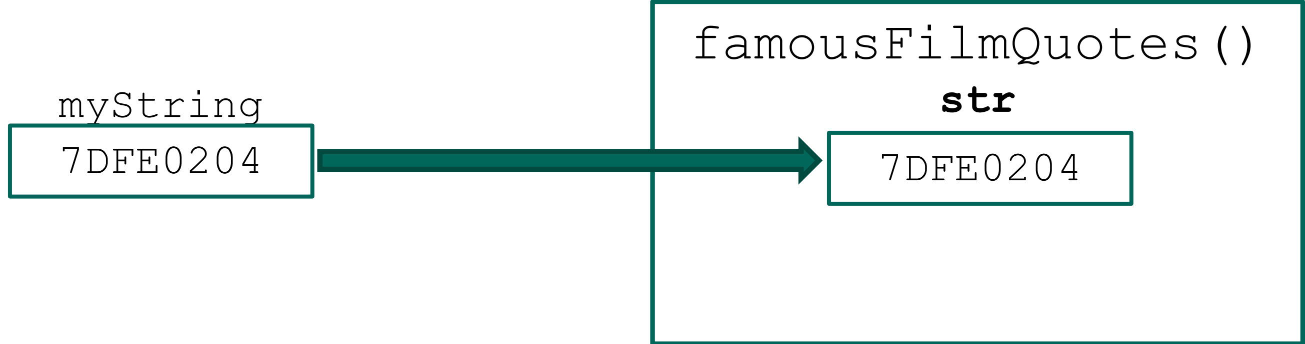
myString
7DFE0204

4.6 Passing a pointer to a function

When you call the function:

```
famousFilmQuotes(myString);
```

you are passing the address stored in `myString` into the function via the variable `str`, which is used in the function declaration

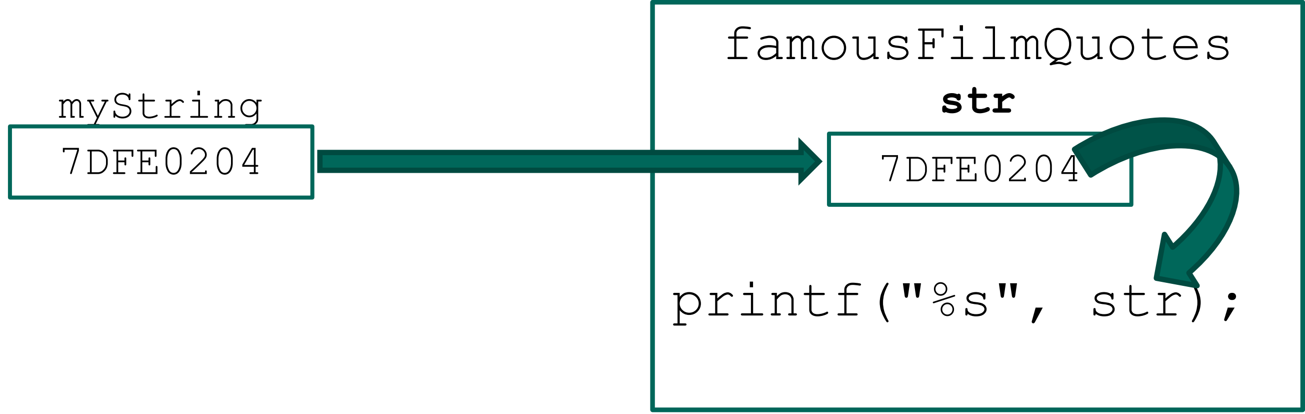


4.6 Passing a pointer to a function

Finally, inside `famousFileQuotes()`, the value stored in `str` is used in the actual `printf()` statement via

```
printf("%s", str);
```

What is important is the fact that *only* the address of the starting "S" is actually transferred at any point; *no other information is ever provided*.



Questions

1. In the earlier example using `nextChar()`, our code should check to make sure that only alphabetical characters have been entered. Therefore, we should use the `isalpha()` function found in the `ctype.h` library first. However, `isalpha()` takes a `char` as its argument (or rather, a `char` inplicitly converted to an `int`), but our function takes a pointer to a char as an argument. How do you convert `ch` so that it can be used by `isalpha()`? (Hint: see the result of the previous Question set.)

2. Note that, rather than set `char *myString =` … in the 'famousFilmQuotes' example and copy the address in `myString` to our function, we could also simply write:

```
famousFilmQuotes("Soylent Green is people!");
```

Why does this work? What are we passing to the function, and how does this fit in with what the function declaration header expects?

Questions

3. When you make the statement

```
char ch[] = "ABC";
```

   how many *total* bytes are actually reserved (assuming a 32-bit PC)?

4. Recall that `printf()` and `scanf()` both take additional string arguments after the initial formatting string. In the case of `scanf()`, we needed to be concerned about whether we passed a primitive data type to `scanf()` e.g. a `char`, `int`, `float`, etc.—in which case we needed to use the `&` operator in front of the variable name—or a reference data type, such as a string of characters, which could be passed directly, without the address operator. Based on your knowledge of pointers, describe why we needed to employ this convention.

4.6 Passing a pointer to a function

- Assume we wish to write a function that swaps the value of two integers. Since data passed directly to a function becomes local inside that function, we cannot pass two variables by value and expect the changes we make to be permanent. Instead, we pass the addresses of the two values.

```
void swap(int*, int*);  // forward declaration

int main(void){
   int x = 1, y=7;
   swap(&x, &y);
}

void swap (int *p, int *q){
   int tmp = *p; // tmp = value pointed to by p
   *p = *q;              // value at *p = value at *q
   *q = tmp;      // value pointed to by q  = tmp
}                 // (=*p original)
```

4.6 Passing a pointer to a function

- When declare a function like

```
swap (int *p, int *q)
```

You specify that two addresses will be passed into this function. Therefore the actual call is:

```
swap(&x,     &y);
```

4.4 Pointers in action – ****Note****

As a rule, always assume the following:

Whenever you see something in the form of a declaration plus assignment, i.e. with the data type being pointed to specified, such as in

```
float *pFloat = …
```

then what appears on the right hand side of the equals sign ***must** be an address*. However, when you see

```
*pFloat  =
```

by itself, without the data type in front, then treat this as equivalent to the value *being pointed to*.

Note that this corresponds to making the following initialization:

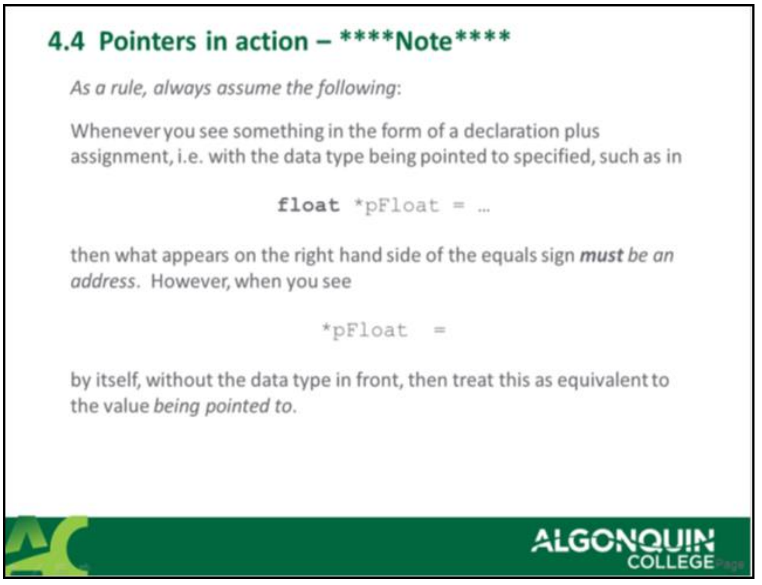
```
int *p = &x       and       int *q = &y
```

Why `&x` and `&y`? Recall the advice from a few slides back. Since this 'looks' like a declaration, you must pass the *address* of `x` and `y`, and *not* the actual contents of `x` and `y` themselves.

`swap(&x,&y);` *not* `swap(x,y);`

Probably not intended

4.6 Passing a pointer to a function


4.4 Pointers in action – ****Note****

As a rule, always assume the following:

Whenever you see something in the form of a declaration plus assignment, i.e. with the data type being pointed to specified, such as in

```
float *pFloat = …
```

then what appears on the right hand side of the equals sign ***must** be an address*. However, when you see

```
*pFloat =
```

by itself, without the data type in front, then treat this as equivalent to the value *being pointed to*.

So, when passing pointers to a function, be sure to use the 'if it looks like a declaration' Rule. If it the assignment you make to a function argument looks like a declaration, like

```
int *p = ...
```

then what is on the RHS *must be an address, not the contents of a variable*.

4.6 Passing a pointer to a function

- The following would also work. Given

  swap (int *p, int *q)

  You could transfer the addresses into temporary variables that held addresses (i.e. pointers). e.g.

  int *pX = &x, *pY = &Y;

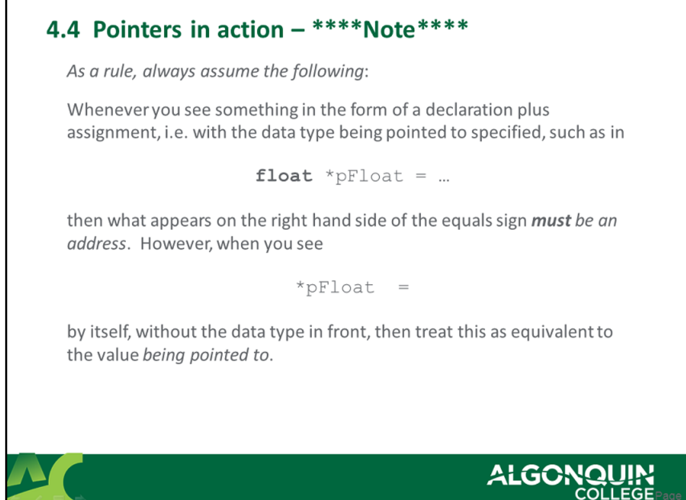
  Then the actual call is:

  swap(pX, py);

  Note that this corresponds to making the following initialization:

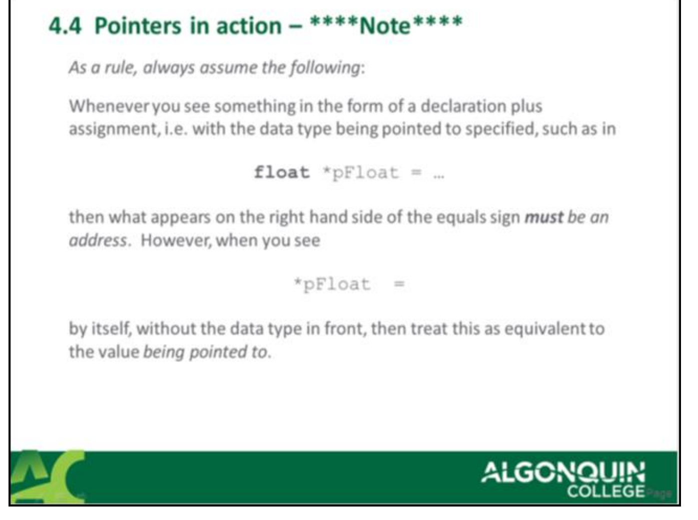
  int *p = pX and **int *q = pY**

  where **pX** and **pY** each store an address. So this approach too is acceptable. But a call to **swap (*pX, *pY)** *is wrong*.

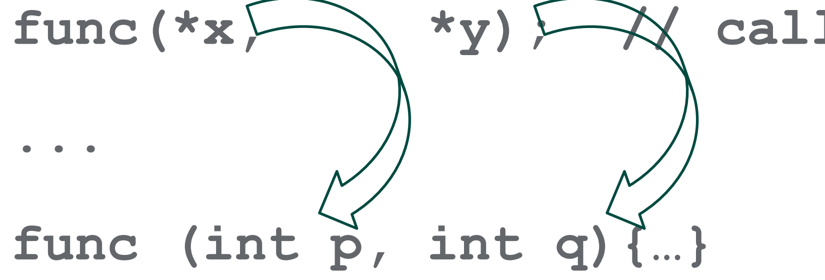


4.6 Passing a pointer to a function

- Consider a different situation. *If* (for some reason) we wanted to pass the value of two `ints` *pointed to* by `*x` and `*y` to a function, then our function call and declaration would look like this:




```
func(*x, *y); // call
...
func (int p, int q){…}     // not (int *p, int *q)
```



and now we are doing something that looks more like a conventional assignment using primitive data types (except the RHS is a pointer):

`int p = *x;` and `int q = *y`

Again, treat the above rule as your guide. `int p` stores a value, not an address. Therefore the function call in NOT `func(&x, &y);`

4.7 C String Operations in the Standard Library

- So using pointers, we can talk to each individual character in the string, exactly as if we where using an index into any array of characters:

```
#include <ctype.h>

char *capitalizeAll(char *iStr){
   char *retString = iStr;  // save starting addr
   while (*iStr != '\0'){
      *iStr = toupper(*iStr);
      iStr++;
   }
   return(retString);       // return start addr
}
```

C provides many such functions for you in the `string.h` standard library.

4.7 C String Operations in the Standard Library

- The standard C (and POSIX compliant) library `string.h` contains a host of common string handling functions. The five most useful are:

Format	Function
`char *strcat(char *s1, const char *s2)`	Appends s2 onto s1, and returns s1
`int strcmp(const char *s1, const char *s2)`	Compares 2 strings and returns an `int` according to whether s1<s2, s1=s2, s1>s2
`char *strcpy(char *s1, const char *s2)`	Moves s2 into s1; it's assumed s1 has enough space to hold s2.
`size_t strlen(const char *s)`	Returns the number of character before \0.*
`char* strstr(const char *s1, const char *s2)`	Searches s1 for the first occurrence of st2.

4.7 C String Operations in the Standard Library

- There is nothing special about the functions in the C library; they are written in human-readable C code and are just what you'd write—if you were a crack C programmer. For example, here's the code behind **strlen()**:

```
size_t strlen(const char *s){
    size_t n;    //size_t declared as ANSI standard
    for (n = 0; *s != '\0'; ++s)
        ++n;
    return n;
}

// Note: could also use a do…while loop
```

ALGONQUIN COLLEGE

4.7 C String Operations in the Standard Library

- As a second example, here's the code behind **`strcpy()`**. `strcpy()` moves the string pointed to by s2 into s1.

```
char *strcpy (char *s1, register const char *s2){
    register char *p = s1;
    while (*p++ = *s2++)
        ;
    return s1;
}
```

Note that the above code is *correct as written*. See if you can figure out how it works.

4.7 C String Operations in the Standard Library

Finally, here's **strcat()**:

```
char *strcat (char *s1, register const char *s2){
   register char *p = s1;

   while (*p)
      p++;  // find the end of the string

   while (*p++ = *s2++)
      ;
   return s1;
}
```

Note that there is no error-checking in any of these functions; the programmer is responsible for allocating sufficient space to accomplish each operation. This leads to a certain vulnerability known as a ***buffer overrun***, which we'll look at later.

4.7 C String Operations in the Standard Library

- Understanding the internals of these functions helps you understand how to use them in instructions. For the following examples, assume:

```
char s1[] ="First we take Manhattan"
char s2[] ="Then we take Berlin"
```

then

Expression	Value Returned
`strlen(s2)`	19
`strlen(s2 + 14)`	5 (i.e. what's left over, to the end of the string)
`strcmp(s1, s2)`	-1 (i.e. "F" < "T" in the 1st char of each string)

Statement	Output
`printf("%s\n", s1+14)`	Manhattan
`strcpy(s1+15, s2+14);`	(Copy the last 5 chars of s2 into position 15 of s1)
`printf("%s\n", s1);`	First we take Merlin

4.7 Returning pointers *from* functions

- Note: in the examples shown thus far, like

```
char *capitalizeAll(char *iStr){
   char *retString = iStr;  // save starting addr
   while (*iStr != '\0'){
      *iStr = toupper(*iStr);
      iStr++;
   }
   return(retString);       // return start addr
}
```

(as well as the `string.h` library functions) an address is passed into the function and returned by the function itself. Thus the function never needed to preserve the address after the function exits; it was supplied by the calling function, and returned to the calling function.

4.7 Returning pointers *from* functions

- If a function like

```
char *myFunction (void);
```

and is *not* returning a pointer to an identifier with a lifespan greater than that of the function itself, then somewhere inside the actual function definition, we would expect to see:

```
char *myFunction (void){
    static char someChar;
    …  // do something to someChar
    return (&someChar);
}
```

If the address returned is not to a `static` (or globally) declared value, then that value may not be accessible when you need it; lacking the proper storage class, the variable 'vanishes' when the function is exited.

4.8 Arrays and Pointers

- As you saw with the array of characters in an example above, arrays and pointers may be treated as essentially equivalent (although *there are* differences: see below). Assume the following declaration:

```
int Ar[] = {0,1,2,3,4,5,6,7,8,9};
```

In C, the name of the array—by itself, without brackets—is equivalent to the address of the first element in the array. Therefore the expression:

```
*Ar
```

literally means: the value pointed to by the address of `Ar`. Since this value is just the first element in the array, it is equivalent to:

```
Ar[0]
```

ALGONQUIN COLLEGE

4.8 Arrays and Pointers

- Alternately, since `Ar` is just equal to the address of the first element of the array, then

`Ar` is equivalent to `&Ar[0]`

Note that both these references refer to the address of a particular location; lacking the dereferencing operator, they have no effect on the value being point to.

4.8 Arrays and Pointers

- When we look at the values stored *in* `Ar[]`,

  `Ar[i]` is equivalent to `*(Ar+i)`

  —they both point to the same place in memory. So the `i`th element of an array is the same thing as what is being pointed to by `Ar` plus an offset. The left side is conventional array notation; the right side uses the address of the array (the value returned by `Ar`) plus an offset, combined in a pointer notation.

4.8 Arrays and Pointers

- For example, say we have following defines/declarations:

```
#define N 100
int A[N], I, *pAr, sum = 0;
```

Say that the starting point of the array, at `A[0]`, is at location 0x300. Then, `A[1]` starts at memory location 0x304; `A[2]` starts at 0x308, etc. The notation

`pAr = A` is equivalent to `pAr = &A[0]`

And then

`*(pAr+1)` points to the same element as `A[1]`

ALGONQUIN COLLEGE

4.8 Arrays and Pointers

- If `A[]` has been assigned values, we can use pointers to loop through and add up all the values:

```
for (pAr = A; pAr < &A[N]; ++pAr)
    sum += *pAr; //add up values pointed to
```

Alternately, we could write:

```
for (i = 0; i < N; ++i)
    sum += *(A + i);
```

Or, equivalently:

```
pAr = A;  // same as pAr = &A[0];
for (i = 0; i  < N; ++1)
    sum += pAr[i];
```

Examples From: *A Book on C 4e*, I. Pohl and A. Kelly, Addison Wesley, 1998

4.8 Arrays and Pointers

And so

`pAr`	points to	`A[0]`	at 0x300
`pAr++`	now points to	`A[1]`	at 0x304
`pAr++`	a second time points to	`A[2]`	at 0x308

etc.

...and `*ptr` successively points to each of the values stored in the array.

Note that C increments the pointer according to the *size of the data type pointed to*; in the above case we have assumed 4-byte `int`s. But if array was declared using `double` data types instead, then we would get:

`pAr`	points to	`A[0]`	at 0x300
`pAr++`	points to	`A[1]`	at **0x308**
`pAr++`	a second time points to	`A[2]`	at **0x310**

etc.

4.8 Arrays and Pointers

- As an example of this, we can print out the difference in locations of two adjacent elements in an array:

```
double a[2], *p, *q;

p = a;          // point to the start of the array
q = ++p;        // now q = &a[1];

printf("%d\n", q - p);                      // 1
printf("%d\n", (int) q - (int) p);          // 8
```

The difference in memory address is equal to the size of the data being stored; the difference between q and p is '1 increment'.

4.8 Arrays and Pointers

Note that if you write:

```
printf("%d\n", p);
```

instead of

```
printf("%d\n", q - p);
```

you will get an error. **p** by itself is just a pointer storing an address; the `%d` type specifier was not designed to hold its value. But **q-p** is a math operation that subtracts the addresses of the two pointers. It's result is an integer number, one that can be handled by `%d`.

*Pointer arithmetic is different; the size of the data pointed to matters.**

*And therefore, you cannot use pointer arithmetic on a pointer to a `void`, since `void` has no 'size'

ALGONQUIN COLLEGE

4.8 Arrays and Pointers

- Note that, since `"abc"` returns an address to the start of the string, we can use it just like a pointer, and apply pointer arithmetic.

```
"abc"[1]      // points to 'b'
*("abc" + 2) // points to 'c'
```

This example serves only to underscore the fact that strings literals are treated as pointers; it is unlikely you'd ever need to use the above code in any serious program.

4.8 Arrays and Pointers

- Just as we have two separate ways to reference pointers

  `int *ptr =…` for definitions—must be set equal to an address; and,

  `*ptr =` to reference what the pointer is pointing *to*

  we also have two ways to use arrays. First, given

```
int Ar[] = {1, 2, 4, 8, 16, 32};
```

  `Ar[]` in this context declares an array, similar to declaring a pointer with `int *ptr`.

  If we wish to initialize a pointer to the above array, we can write:

```
int *ptr = Ar;
```

  Here, `Ar` (without the `[]`) returns 'the address of the first element of the array'.

4.8 Arrays and Pointers

- To use the array *after* declaration, we use the conventional method to access the `i`th element of an array by referencing:

```
Ar[i]
```

which could also be accessed via

```
*(ptr + i)
```

Note that, while `Ar[]` is used in declarations, `Ar[i]` returns the element that is being pointed to (in exactly the same way that `int *ptr` is used in declarations and needs to be assigned an address, while `*ptr` is literally 'what is being pointed to'.)

4.8 Arrays and Pointers

- As usual, we can break the process of initializing a pointer into a two-step process:

```
int *ptr;   //create pointer to an int.
ptr = Ar;   //Ar returns the address of 1st int in Ar.
```

Or, equivalently, for the second step, we can use:

```
ptr=&Ar[0]; //&Ar[0] is the address of 1st int in Ar.
```

Notice that:

```
ptr = &Ar;   //Wrong!!!
```

`&Ar` is not an address that points to an `int`

signals an error. Why? Because, while `&Ar` *is an address* to our array, it is not a pointer to an `int`, which is what `ptr` expects. So `Ar` returns the correct data type expected by the compiler, while `&Ar` does not.

4.8 Arrays and Pointers

- Here's the second way to refer to our array, `Ar`.

  When we write the square brackets with the array, we must do so only as part of a declaration. For example, in

```
int Ar[] = {…};
```

  `int Ar[]` is similar to

```
int *ptr = …;
```

  in the sense that it should always appear on the LHS of the expression. If you see something like:

```
int *ptr = Ar[];   //This is Wrong!!!
```

  Outside of a declaration, `Ar[]` used by itself in an expression is wrong. There is, of course, nothing wrong with:

```
*ptr = Ar[0];     //since Ar[0] is an int
```

4.8 Arrays and Pointers

- Alternately, we can write our array/pointer declaration/assignment all in one go:

```
int Ar[] = {1, 2, 4, 8, 16, 32}, *pAr = Ar;
```

But when the time comes to actually use the array to talk to the elements, `Ar[i]` and `*pAr` refer to the actual data being addressed:

```
if (Ar[2] == 2* (*(pAr +1))) {
    …
}
```

So arrays, like pointers, have two slightly different forms with similar syntax: one form occurs during initialization and has looks like `int *ptr = …` or `int Ar[] = …`; the second form is used to access the actual value *being pointed at* and has the form `*ptr` or `Ar[i]`.

ALGONQUIN COLLEGE

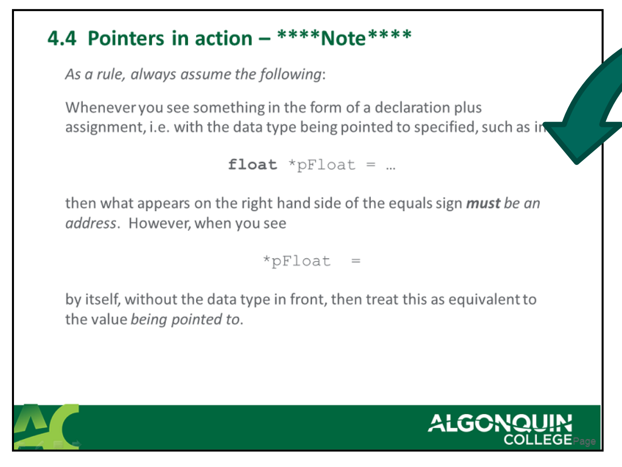
4.8 Arrays and Pointers

- The same relationship between pointers and arrays applies when passing an array as an argument to a function. Assume the function `sort()` takes as an argument an array of integers to be sorted. We can pass the address of that array using an argument list declared as either:

```
void sort (int *x) {…}
```

or as

```
void sort (int x[]) {…}
```


4.4 Pointers in action – ****Note****

As a rule, always assume the following:

Whenever you see something in the form of a declaration plus assignment, i.e. with the data type being pointed to specified, such as in

```
float *pFloat = …
```

then what appears on the right hand side of the equals sign ***must*** *be an address*. However, when you see

```
*pFloat =
```

by itself, without the data type in front, then treat this as equivalent to the value *being pointed to.*

The same rules apply when passing arrays to functions as applied when passing pointers to functions

4.8 Arrays and Pointers

As with `int *x`, `int x[]` expects to 'see' the address of the first integer in an array or, alternately, the address of a pointer to an `int`. So, given the following declarations

```
int Ar[] = {-2, 5, 6, -10, -3, 1}, *ptr = Ar;
```

when using a function declared as:

```
void sort (int x[]) {…}
   ...
```

any of the following is allowed:

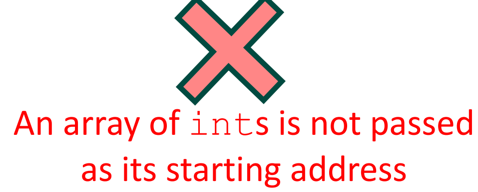
```
sort(Ar);
sort(&Ar[0]);
sort(ptr);
```

because `int x[]` expects to be set equal to an address, and each passes an address into the argument list.

4.8 Arrays and Pointers

However, note that

```
sort({-2, 5, 6, -10, -3, 1});
```



is not allowed. While "abc" is literally equal to the first address of an array of chars, the same doesn't apply to `{-2, 5, 6, -10, -3, 1}`; *strings are a special case*.

4.8 Arrays and Pointers—differences

- Note that, despite appearances, arrays and pointers are not *quite* the same thing:

  - pointers store an address; arrays store a collection of data;
  - A pointer is 'agnostic' in that, technically, it doesn't know what it is pointing too. At best, *you* tell the compiler 'this is a pointer to a ….', but the pointer itself only contains an address, nothing more. By contrast, an array has some number of identically-typed values, as well as addresses for each value, from the first element of the array to the last;
  - pointers allow for pointer arithmetic; arrays can allow for arithmetic on the indices that 'point' into the array, but it's not quite the same.
  - In actual use, an array is often *implicitly* converted to a pointer to the first element of the array. However, this is different from the situation in which a pointer is *explicitly* created. Thus, you can use `*Ar` in place of `Ar[0];` that doesn't mean you've actually reserved space internally for a pointer.

4.8 Arrays and Pointers—differences

Two of the biggest differences between *the use of* pointers and arrays are especially important:

1. A pointer can take different addresses as values, but an array element will always have the same address as it started with. An array, `Ar[]` *is assumed to be a **constant** pointer*. Thus if `ptr` is a pointer, then

```
ptr = Ar;        // is legal, but…
Ar = ptr;        // …is not.
```

Ar is a constant; it cannot be assigned a new value

4.8 Arrays and Pointers—differences

2. When "an array is passed to a function", *only a pointer to the array is actually passed. Just because you declare the function as*

```
void swap (int x[], int y[]) {…}
```

doesn't mean you're passing the contents of the arrays into the function. Despite appearances, ONLY THE ADDRESSES ARE PASSED.

4.8 Arrays and Pointers

- This last point has important consequences for anyone used to the world of objects. When passing a pointer to a function, remember:

A pointer is an address,

and only an address;

unlike an object, it does not contain

any additional information.

4.8 Arrays and Pointers

- This fact is most obvious when we use the `sizeof()` operator on an array that has been passed as a function argument, as the following code demonstrates:

```
#include <stdio.h>

void PassArrayToFunction(char []);      // Note function prototype

int main(void){
   char Ar[] = "abcdefghijklm";         // 13 chars plus '\0'
                                        // this returns 14
   printf("Array size before function call = %zu\n", sizeof(Ar));
   PassArrayToFunction(Ar);
}

void PassArrayToFunction(char ar[]){
                                        // this returns 4
   printf("Size of array inside function = %zu\n", sizeof(ar));
   printf("Output is: %s\n", ar);    // still outputs abcdefghijklm
}
```

4.9 lvalues and rvalues

- Note that during a declaration, `int Ar[3]` can be used to declare an array of *3* elements; *after* it is declared, `Ar[2]` refers to the 3rd element of that array.

- `Ar` by itself *always* refers to the first address of an array. Therefore, you cannot write

```
Ar = …;
```

WRONG! Ar is an rvalue

`Ar` is an ***rvalue***; it appears only on the right hand side of the equation, *after* the equals sign. (Furthermore, it's constant…)

On the other hand, when you declare

```
int *ptr;
```

Both `ptr` *and* `*ptr` can be used as either an *rvalue* or an ***lvalue***; they can appear on either the left or the right side of the equals sign.

4.9 lvalues and rvalues

- So, given

```
int Ar[] = {1,2,3}, *pAr;
```

this is true:

```
pAr = Ar;    //CORRECT
```

since it says that the address of the first element in the array (an rvalue) is assigned to the pointer (used here as an lvalue)—the address of the first element of the array. But this is not true:

```
Ar = pAr;
```

WRONG! Ar is an rvalue

since, not only is `Ar` technically an rvalue, but it is *always* just the address of the first element of the array, and can't be re-assigned.

ALGONQUIN COLLEGE

4.9 lvalues and rvalues

- Note that when you declare an array, you must give an indication of its size. So this:

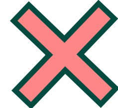
```
int Ar[] = {4,5,6};
```

is allowed, since the size is implicitly defined in the declaration (=3). However, while this

```
int Ar[3];
```

is allowed, you cannot then go on an attempt to assign elements to the array after the assignment, like this:

```
Ar[]= {4,5,6};
```



No late assignments

C only allows arrays to be initialized at the time of declaration. Again, an array of chars is a special case.

ALGONQUIN COLLEGE

4.9 `lvalues` and `rvalues`

- *In general, you cannot make an assignment after a declaration*. So given

```
int Ar[], *pAr;
```

the following trigger errors:

```
Ar[] = {1,2,3};
pAr  = {1,2,3};
```

Again, the one exception to this rule involves chars. If our declaration is

```
char *pAr;
```

then, in this one particular case

```
pAr  = "abc";      // OKAY!
```

but *only* because `"abc"` returns an address to start of the string.

Note: Array declarations in C versus Java ⌘

In Java, when you declare an array without initializing it, the size of the array appears on the *right hand side* of the statement:

```
array_type  array_name[] = new  array_type[size];   //Java
```

e.g.

```
int monthLength[] = new int[12];   //Java
```

But in C, an array declared with an explicit size always appears in the brackets on the *left hand side*, like this:

```
int monthLength[12];  // C
```

Why? In C, an array is a collection of identically-typed data; this is a simple declaration that reserves some number of `int`s, `char`s, `float`s, etc. In Java, an array is an object, and so the declaration takes on the form of a full-blown array declaration, complete with duplicate `int`s on each side of the equals sign.

Summary of Arrays and Pointers in use

The three most important things to remember with regard to actual pointer use are:

1. `Ar` and `"abc"` return the address of the first element in the array (of `int`s, `char`s, etc.) Therefore, they are always rvalues. They can be assigned to a pointer, but they cannot always be treated like pointers, since they cannot be reassigned a new value.
2. A pointer to an array of chars is a special case; it can be assigned a string after declaration. Hence `pAr = "abc"` is allowed, but `Ar[]="abc"` is not. This applies to pointers and arrays inside structures as well, as we'll see shortly.
3. All other 'late' assignments are prohibited, unless they are done piecewise, i.e. `Ar[0] = 1; Ar[1] = 2; Ar[2] = 4;` etc.

Example 14 : Palindromes

Program Description:

A palindrome is a string of characters or numbers that reads the same backwards as forwards. Thus "ABCBA" is palindrome, as is "128821", but "PAPA" is not. The following program passes a string into a function that determines whether the string is a palindrome or not. (In general, palindromes may contain spaces. This code was not built with this possibility in mind, but with a small modification, it could be)

Example 14: Palindromes

```
#include <stdio.h>

int isPalindrome(char[]);

int main(void){
   char str[80] = {'\0'};
   gets(str);
   printf("%s%s%s%s%s", "The string ", str, " is",
        isPalindrome(str)?"":" not", " a palindrome\n");
}

int isPalindrome(char *p){
   unsigned int len=0, ctr=0, palindrome;
   char *start = p;            // save start of the array
   while (*p++)               // increment until '\0' is reached
      len++;                  // len will equal number of input chars
                              // check string from each end
   p -= 2;                    // p now points to last char
   while (++ctr <= len/2)    // loop until midstring
      if (!(palindrome = (*start++ == *p--))) break; //if different
   return (palindrome);
}
```

Example 14: Palindromes

- The code for the palindrome function starts by reading in the start of the array as a pointer p and initializing variables, including the starting address

```
int isPalindrome(char *p){
unsigned int len=0, ctr=0, palindrome;
char *start = p;
```

The code loops from the start of the array to the end and counts characters

```
while (*p++)
   len++;
```

We want the pointer to point to the last character in the string:

```
p -= 2;                 // Why do we need to subtract 2???
```

Start comparing the values pointed to at either end of the string, move toward the middle, and break if the two characters pointed to are not equal.

```
while (++ctr <= len/2)  // loop until midstring
   if (!(palindrome = (*start++ == *p--))) break;
return (palindrome);
```

Questions

1. The following code shows that pointers and arrays are not always equivalent. Given the following function that takes in an address, increments it, and returns it to the calling function:

```
#include <stdio.h>
char* nextAddr(char* s){
     s++;
     return s;
}
```

The following function prints out correctly

```
int main(void){
  char *theString = "Jessica";  // this works; no error
  printf("now pointing to address %p \n",theString);
  theString = nextAddr(theString);
  return 0;
}
```

Questions

But the following code flags an error at the line indicated below:

```
int main(void){
  char theString[] = "Jessica";  // this flags an error
  printf("now pointing to address %p \n",theString);
  theString = nextAddr(theString); //error occurs here
  return 0;
}
```

The only difference between the two programs is that the first contains a pointer declaration to a string, while the second contains an array declaration. Aside from that, the function `nextAddr()` performs the same operation. Explain why one works, and the other does not.

Example taken from: http://www.physicsforums.com/showthread.php?t=37581

ALGONQUIN COLLEGE

4.10 Using `ato...()`

- C provides a family of functions in `<stdlib.h>` that are useful for converting a string value to a `float`, `long`, or `int` (i.e. these are essentially C's version of `.nextFloat, .nextLong, .nextInt` in Java.) Each function has the form `ato...()` and converts a string of alphabetical characters to integers:

```
int   atoi(const char *s) converts string to integer
float atof(const char *s) converts string to float
Long  atol(const char *s) converts string to long
```

Usage:

```
int radius;
char input[80] = {'\0'};
scanf("%s\n", input);
radius = atoi(input); // takes address as arg
printf("area = %f\n", 3.1415*radius*radius);
```

4.11 2-D Arrays

- 2-Dimensional (and higher) arrays are possible in C. A 2-D declaration would look like this:

```
int ar[4][5];     // a 2D array; the first index is the row;
                  // the second is the column
```

col →

row ↓

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	ar[0][0]	ar[0][1]	ar[0][2]	ar[0][3]	ar[0][4]
Row 1	ar[1][0]	ar[1][1]	ar[1][2]	ar[1][3]	ar[1][4]
Row 2	ar[2][0]	ar[2][1]	ar[2][2]	ar[2][3]	ar[2][4]
Row 3	ar[3][0]	ar[3][1]	ar[3][2]	ar[3][3]	ar[3][4]

To print this out:

```
for (row = 0; row < ROWMAX; row++){
    printf("\n");
    for (col = 0; col < COLMAX; col++)
        printf("ar[%d][%d]\t", row, col);
}
```

4.11 2-D Arrays

- To initialize an array, use one of the following methods. Each of the following declarations assume an array of 2 rows by 3 columns.

```
int ar[2][3] = {0};        // initializes the entire array to 0

int ar[2][3] = {};         // same effect; everything set to 0

int ar[2][3] = {{0}}       // same; removes -Wall warning

int ar[2][3] = {{0,1,2}, {3,4,5}};    // set separately
                                      // 2 rows, 3 columns

int ar[2][3] = {0,1,2,3,4,5};         // same as above

int ar[][3]  = {0,1,2,3,4,5};         // OK; rows must be 2
                                      // since 6 elements total

int ar[2][3] = {7}; // sets array to {7,0,0,0,0,0,0} not {7,7,7…
```

ALGONQUIN COLLEGE

4.11 2-D Arrays

- Internally, 2D arrays are stored as a sequential series of numbers. Thus

```
ar[3][5] = {{0,1,2,3,4},{5,6,7,8,9},{10,11,12,13,14}};
```

which is often written as:

```
ar[3][5] = {
                {0,1,2,3,4},
                {5,6,7,8,9},
                {10,11,12,13,14}
            };
```

to emphasize the structure of the array in rows and columns, is stored internally as:

```
ar[3][5] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14};
```

4.11 2-D Arrays

- C does not actually allow for true two-dimensional arrays. A 2D array is best though of as an array of arrays. For example

```
ar[2][3]
```

an array of 2 rows by 3 columns, is really

```
(ar[2])[3]
```

which is best thought of as a 1D array containing 2 elements, with each element consisting of 3 sub-elements. Thus if

```
ar[2][3] = {{0,1,2},{3,4,5}};
```

then

`ar[0]` corresponds to `{0,1,2}` and `ar[1]` corresponds to `{3,4,5}`

4.11 2-D Arrays

- Indeed, we can reference the memory at the start of each row using the notation above. To output the address stored in a pointer, use the %p format specifier. Thus in the program:

```
#include <stdio.h>

int main(void){
   int Ar[2][4] = {{1,2,3,4}, {9,8,7,6}};
   printf("Start of first row equals %p\n", Ar[0]);
   printf("Start of second row equals %p\n", Ar[1]);
   printf("Separation equals %d\n", Ar[1]-Ar[0]);
}
```

The output is:

```
Start of first row equals 0xbfdd9490
Start of second row equals 0xbfdd94a0
Separation equals 4
```

4.11 2-D Arrays

- Note that pointer arithmetic applies here. The separation between the start of the first row and the start of the second is 4—the number of elements separating row 1 from row 0. We can display the structure of this array as:

0xbfdd9490	0xbfdd9494	0xbfdd9498	0xbfdd949c
0x00000001	**0x00000002**	**0x00000003**	**0x00000004**

0xbfdd94a0	0xbfdd94a4	0xbfdd94a8	0xbfdd94ac
0x00000009	**0x00000008**	**0x00000007**	**0x00000006**

Thus the separation between **0xbfdd94a0** and **0xbfdd9490** is

$$\frac{\texttt{0xbfdd94a0 - 0xbfdd9490}}{\texttt{sizeof(column)}} = 4$$

4.11 2-D Arrays

- We can reference the same information using pointers. A 2D array is always represented internally, but once we apply pointer notation, internal conversions are used that allow us to treat a 2D array as an array of pointers

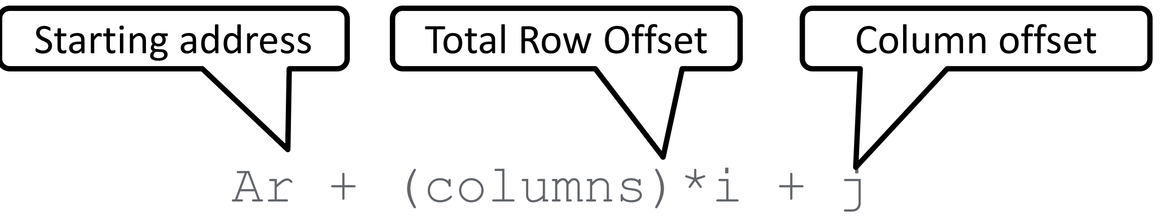
```
#include <stdio.h>

int main(void){
   int Ar[2][4] = {{1,2,3,4}, {9,8,7,6}};
   printf("Start of first row equals %p\n", *Ar);
   printf("Start of second row equals %p\n", *(Ar + 1));
   printf("Separation equals %d\n", *(Ar + 1)- *Ar);
}
```

The output is:

```
Start of first row equals 0xbfdd9490
Start of second row equals 0xbfdd94a0
Separation equals 4
```

4.11 2-D Arrays

- As before, `Ar` points to the address of the start of the array (which should also correspond to the address of the first row,
- So what both `Ar[i]` and `*(Ar + i)` point to the addresses at the start of the ith row;
- `Ar[i][j]` and `*(Ar + i)[j]` point to the element stored in the jth column of the ith row.
- As always, `&Ar[i][j]` returns the *address* of the jth column of the ith row, as does `&(*(Ar + i)[j])`, which is just `(Ar + i)[j]`—since the `&` and `*` symbols cancel out. Another way to think about this last form is that the address of the element located at the jth column of the ith row is:



4.11 2-D Arrays

- The same reasoning applies when using the `sizeof()` operator. For an array declared as `int ar[5][6]`, `ar` *refers to the array*; so

```
sizeof(ar) == 120(= 5 X 6 X 4 bytes/int)
```

In contrast, `*ar` refers to what the array points to, i.e. a row, so

```
sizeof(*ar)== 24    //= size of row = 6 cols X 4 bytes/int
sizeof(*ar+3)==24   //= size of another row
```

But pointer arithmetic plays a role in the examples above. Note that:

```
(int)(*(ar+3)) - (int)(*ar)) // = 72
                                //i.e. 3 * 24 = 72 bytes
```

ALGONQUIN COLLEGE

4.11 2-D Arrays

- As we've seen, pointers and arrays are very nearly equivalent. Given an array, `ar[r][c]`, the following are equivalent statements:

```
*(ar[r] + c)          // points to the array's r^th row, plus offset of c

(*(ar + r))[c]        // pointer arithmetic; start at ar in memory and add
                      // (r-1 rows) * (cols per row). Then add the offset c.

*((*(a + i)) + j)     // The value pointed to at the i^th row + j^th column

*(&a[0][0] + c*i + j)    // Offset plus column width time
                         // number of rows, plus final offset in columns
```

4.12 Arrays of Strings

- We can also use a 2D array to store an array of strings. Note that the size of the column element must be at least the size of the longest word in the array:

```
char st[][8]={"this", "is", "an", "array", "of", "strings"};
```

So in a real sense, an array of arrays of chars is represented internally as an row- by column-sized array of characters:

col →

row ↓

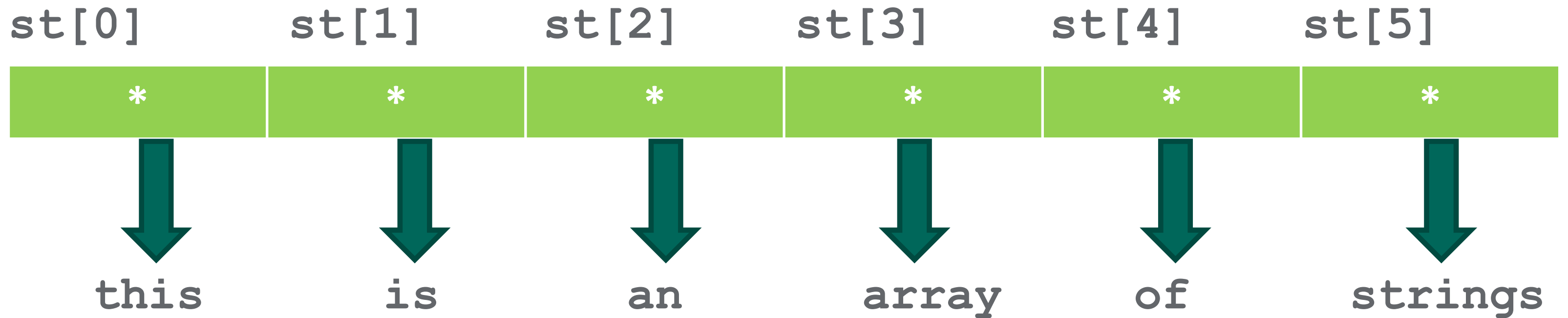
t	h	i	s	\0			
i	s	\0					
a	n	\0					
a	r	r	a	y	\0		
o	f	\0					
s	t	r	i	n	g	s	\0

4.12 Arrays of Strings

- Alternately, with pointer notation, we can point to individual strings within an array of strings. For example, given

```
char *st[]={"this", "is", "an", "array", "of", "strings"};
```

this is represented internally as an array of six pointers to `char`s. Thus:



4.12 Arrays of Strings

- To output this array, you could write:

```
for (ctr=0; ctr < 6; ctr++)
    printf("%s ", st[ctr]);
```

or

```
for (ctr=0; ctr < 6; ctr++)
    printf("%s ", *(st + ctr));    //same as st[ctr]
```

and this will print the words in `st[]`. Note that each string is separately terminated with a '\0'.

4.12 Arrays of Strings

- Note that, while they produce the same output, the two definitions

```
char ar[2][15] = {"abc", "a is for apple"};
char* ptr[2]   = {"abc", "a is for apple"};
```

store their information differently. `ar` is a true 2D array, and the assignment above is the equivalent of:

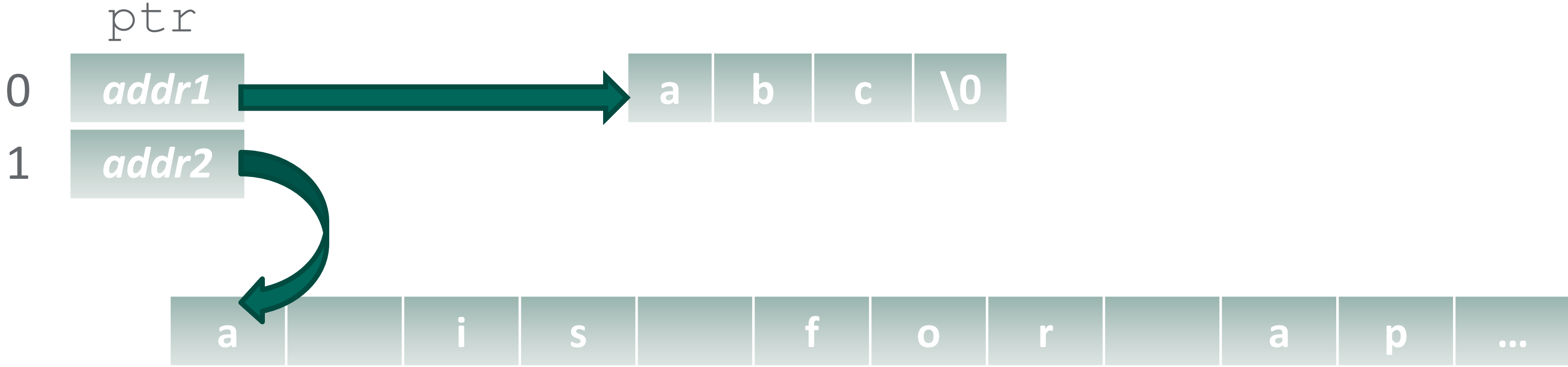
```
{{'a', 'b', 'c', '\0'}, {'a', ' ', 'i', 's', ' ',…,'\0'}}
```

where the first row, `ar[0]`, is filled out with 0's, since `{'a', 'b', 'c', '\0'}` has fewer than 15 characters in it. So `ar` looks like this:

a	b	c	\0	0	0	0	0	0	0	0	0	0	0	0
a		i	s		f	o	r		a	p	p	l	e	\0

4.12 Arrays of Strings

- By contrast, `ptr` is an array of pointers to chars, i.e.



An array of pointers used in this way, holding strings of various lengths rather than a fixed size, is called a ***ragged array***. Note that a ragged has more memory overhead initially: it requires that pointers are created to the start of each string. But this saves space overall, since there's no need to create a full 2D array, which may be filled with a lot of empty, unused space.

For holding a large array of strings, pointers make far more efficient use of space than arrays: a ragged array is superior to a fixed array.

4.13 Passing information at the command line

- An array of strings is passed to every program via the `main()` parameter list. We often suppress the arguments to `main` by using `void` as the argument (as in `int main(void)`). But the full version of is:

```
int main(int argc, char *argv[])
```

where

`argc` refers to the number of strings passed in the array

`*argv[]` is an array of pointers to the strings

4.13 Passing information at the command line ⌘

- Note that Java only passes one `String` argument because, in Java, a `String` is an object data type. And the `String` object has a `length` property, hence no need to pass an `argc` value.

  Note also that in Java you can write either:

```
public static void main (String args[]){…}
```

or

```
public static void main (String []args) {…}
```

and Java doesn't care. In C, the brackets must come after the argument variable name itself.

4.13 Passing information at the command line

- These arguments are used to return an array of string data from the command line. For example, when you type

```
gcc -o outputfilename myfile.c -ansi -pedantic -Wall
```

you are passing 7 separate arguments to `gcc` (including the `gcc` command itself). Hence `argc` = 7, and

```
argv[0] = "gcc"
argv[1] = "-o"
argv[2] = "outputfilename"
argv[3] = "myfile.c"
argv[4] = "-ansi"
argv[5] = "-pendantic"
argv[6] = "-Wall"
```

4.13 Passing information at the command line

- As an example, the following code echoes the input in reverse order:

```
#include <stdio.h>
int main(int argc, char *argv[]){
    int i;
    for (i = argc; i > 1; i--)
        printf("%s ", argv[i-1]);
    print("\n");
}
```

Typing this at the command line

```
$echoback This sentence is backwards
```

yields:

```
backwards is sentence This
```

ALGONQUIN COLLEGE

4.13 Passing information at the command line

Another version of this, using swap() to swap the actual pointers to the strings in the argument list, is:

```
void swap(char* p, char* q){
      char* tmp = p;       // holds pointer to char
      p = q;               // copy q's address to p
      q = tmp;             // copy tmp address to q
}

int main(int argc, char *argv[]){
   int i;

 //swap last two pointers,ie last two words in sequence
   swap(argv[argc-2], argv[argc-1]);
   for (i = 1; i < argc; i++)
      printf("%s ", argv[i]);
   printf("\n");
}
```

4.13 Passing information at the command line

- This has the effect of swapping the pointers to the last two words in a sentence. When the array of pointers is printed out in order, these two words appear out of sequence, e.g. if the program is compiled as `YodaTalk.c`, then we get the following

```
$YodaTalk Use the force
Use force the

$YodaTalk Nouns before verbs we put
Nouns before verbs put we

$YodaTalk Good english speaks Yoda
Good english Yoda speaks
```

Example 15 : The Input Usage Function

"command_name"
"first_argument"
"second_argument"
"third_argument"
NULL
argv

Program Description:

Since almost all C programs are executed at the command line, and some of those programs will take input arguments, it's useful to have a function that checks to make sure the correct number of arguments was entered. This short function shows how to check that the number of input arguments is correct, and if it isn't, outputs a message explaining correct usage. This function will be used in future programs.

Example 15: The Input Usage Function

```
#include <stdio.h>

void printUsage(char *);

int main(int argc, char **argv){
   if (argc != 3){      // assume we require 3 arguments
      printUsage(argv[0]); // Address of first string
      exit(1);             // Exit program and try again.
   }
   ...
}

void printUsage(char *pgm_name){
   printf("\n%s%s%s\n\n%s%s\n\n",
      "Usage:  ", pgm_name, "  infile  outfile",
      "The contents of infile will be double-spaced ",
      "and written to outfile");
}
```

ALGONQUIN COLLEGE

Example 15: The Input Usage Function

- This implementation assumes that the program requires the user to enter three arguments at the command line. The first argument, `arg[0]`, will, of course, be the name of the program itself. If the user didn't enter three value, then `main()` calls the input usage function

```
void printUsage(char *);

int main(int argc, char **argv){
    if (argc != 3){      // assume we require 3 arguments
        printUsage(argv[0]); // Address of program name
        exit(1);             // Exit program and try again.
    }
    ...
```

Example 15: The Input Usage Function

- The `printUsage()` function provides the 'error message' that tells the user what the correct input is. The function takes the address of the first element of the array, which is the name of the program itself:

```
void printUsage(char *pgm_name){
```

Then comes the message itself. Since `pgm_name` contains the starting address of the name of the program, it can be passed directly to `printf()`. (Remember: `printf()` uses pointers!). The message printed can, of course, be anything.

```
    printf("\n%s%s%s\n\n%s%s\n\n",
    "Usage:  ", pgm_name, "  infile  outfile",
    "The contents of infile will be double-spaced\n",
    "and written to outfile");
}
```

ALGONQUIN COLLEGE

Example 15: The Input Usage Function

- The output is:

```
$ Usage:  double_space  infile  outfile
  The contents of infile will be double-spaced
  and written to outfile
```

Note that, assuming the number of arguments passed at the command line is correct, `main()` could use the remaining arguments to execute code based on the value input:

```
//search for "infile" in arg[1] using string.h ftns
if (strstr(arg[1], "infile") != NULL){
   ...
}
```

4.14 Passing a Function as an Argument to a Function

- C allows the programmer to pass one function into another as a parameter of the second function. For example, say we wish to know the sum of the square of some function, *f(x)* over. Mathematically, this is:

$$\sum_{i=1}^{n} f^2(n)$$

Where we have yet to determine which function we will be summing over. We can write this programmatically using two functions. The first is the sum of the squares for any particular function:

```
double sumOfSquares(double f(double x), int n){
    double sum = 0.0;
    for (int i=1; i <= n; ++i)
        sum += f(i) * f(i);
    return sum;
}
```

4.14 Passing a Function as an Argument to a Function

- The second function, the function to be passed, can be any function that takes in a double and returns a double. If we want to calculate:

$$\sum_{x=1}^{n} \frac{1}{n^2}$$

then our function to pass into `sumOfSquares()` would be:

```
double f(double x){
    return (1.0/x);
}
```

ALGONQUIN COLLEGE

4.14 Passing a Function as an Argument to a Function

- Note that we can enter any function into `sumOfSquares()`, e.g.

```
#include <math.h>
double g(double x){
   return (sin(x * 2* M_PI));
}
```

So now calling

```
double sum = sumOfSquares(g(1), 10};
```

returns

$$\sum_{i=1}^{10} sin^2(n * 2\pi)$$

4.14 Passing a Function as an Argument to a Function

- The function passed as an argument to a function is passed as a pointer, and so the function definition can be written as:

```
double sumOfSquares(double (*f)(double x), int n){
      ...
}
```

This is a seldom-used but powerful feature of C. The fact that functions can be treated as pointers means that you can create an array of pointers to a 'stack' of functions.

In Java, you *can* do something similar, but it involves, as you might expect, passing one object method into another. C's version is direct, more elegant, and computationally *much* faster.

4.14 Passing a Function as an Argument to a Function

Just as in a *function prototype* like:

```
int myFunction(int mydata);
```

the variable `mydata` is an unnecessary placeholder—we can do without it, so long as we mention the data type passed to the function—so also, in a function prototype that includes a function as an argument, like:

```
double sumOfSquares(double (*f)(double x), int n);
```

the function name `f()` is just a placeholder for whatever function we pass into `sumOfSquares()`. Any one of the following works equally well:

```
double sumOfSquares(double f(double), int);
double sumOfSquares(double (*f)(double), int);
double sumOfSquares(double (*)(double x), int n);
double sumOfSquares(double (*)(double), int n);
```

4.15 Unscrambling Declarations in C

- Reading function prototypes can be a tricky business in C. As a general rule, you should:

  1. Start at the leftmost variable name in the declaration
  2. Do what's in brackets first
  3. If the token to the right of the variable name is [] or (), then it comes next, otherwise
  4. Read from right to left.

For example:

```
const int * grape       // grape is a pointer to a constant integer
int const * apple       // apple is a pointer to a constant integer
int * const lemon       // lemon is a constant pointer to an integer
```

4.15 Unscrambling Declarations in C

Recall that to create a constant pointer to a constant integer, you must use:

```
const int * const orange;
```

Alternately, you can use:

```
int const * const orange;
```

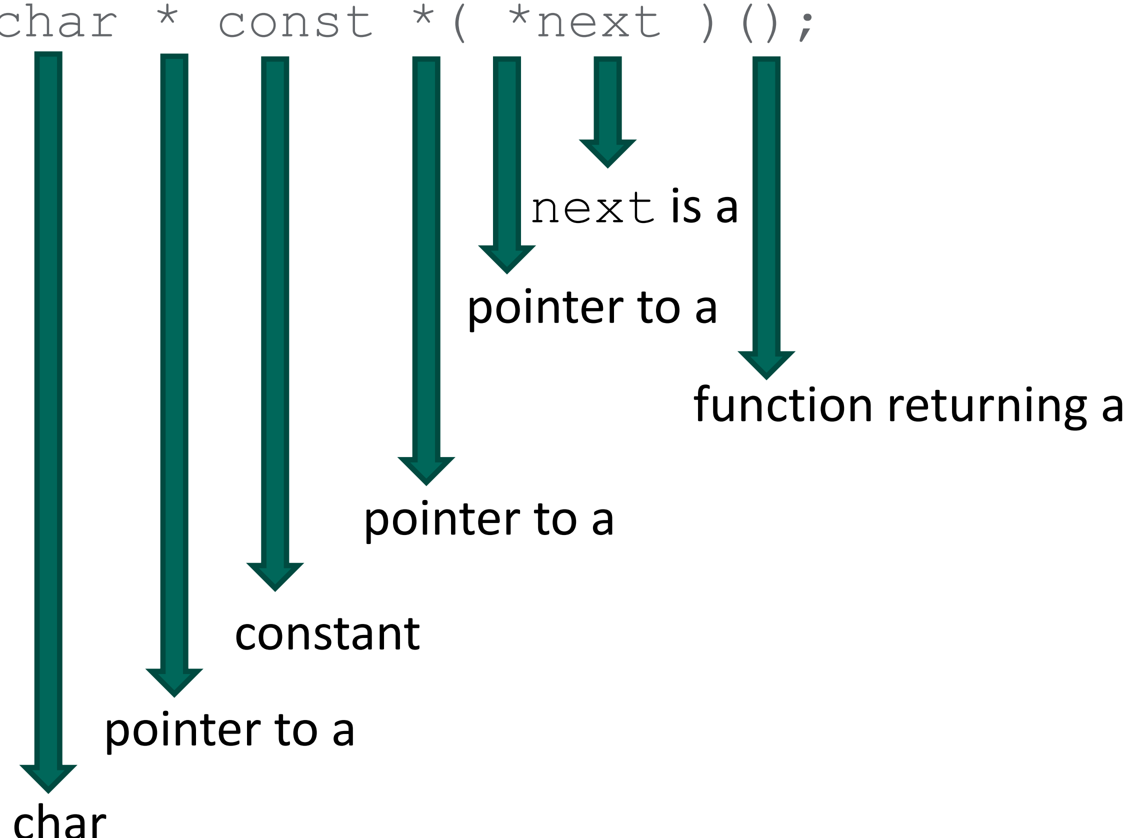
Which is exactly the same, and follows the 'read it right-to-left' rule more consistently, and is more intuitive:

"`orange` is a constant pointer to a constant integer"

4.15 Unscrambling Declarations in C

Therefore:

reads:



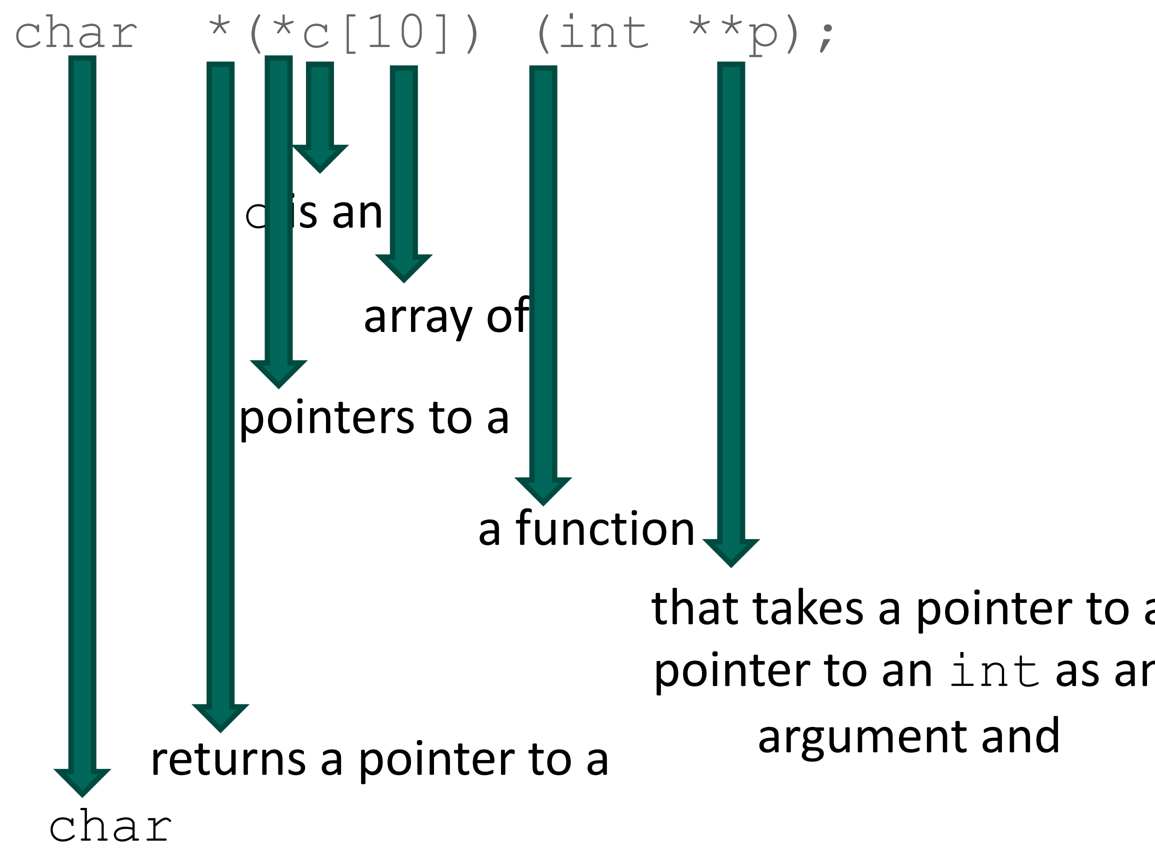
This is an actual declaration adapted from a `telnet` program: see *Expert C Programm: Deep C Secrets*, P. v.d. Linden, Prentice Hall, 1994, pg. 66.)

4.15 Unscrambling Declarations in C

Another example:

```
char  *(*c[10]) (int **p);
```

reads:



So, "c is an array [of] pointers to a function that takes a pointer to a pointer to an int as an argument and returns a pointer to a char."

4.15 Unscrambling Declarations in C

Certain combinations of identifiers/reserved words are not possible in C. Using the above rules, we can see that the following combinations are not feasible:

```
foo()()      // foo is a function
             // returned by a function?

foo()[]      // foo is a function
             // that returns an array?

foo[]()      // foo is an array
             // of functions?
```

Note that with brackets, pointers, and forethought, the following *are* possible:

```
int (* foo())()
int (* foo())[]
int (*foo[])()
```

Useful Web Sites for Studying up on Pointers

Basic questions involving pointers and functions in C, with answers

http://www.psut.edu.jo/sites/yahiah/yahia/Practice%20Questions%20for%20C%20programming.pdf

Questions Focusing exclusively of tricky pointer issues, with answers:
(some may contain C++ questions, so not always ideal for studying with)

http://www.cquestions.com/2012/02/c-pointers-questions.html

http://www.examveda.com/programming-aptitude/c-programming/pointer/001001/

http://www.ittestpapers.com/articles/programming-question-and-answers-in-c-language---test-your-c-skills.html

Pointers are used in many areas of C programming, and we haven't covered everything yet. So these web sites may contain questions involving topics to be taught later in the course, like structures, enums, and files.