ATSC 212 - C

C

ATSC 212 - C

The C programming language was originally designed and implemented by Dennis Ritchie to be a flexible and versatile programming language that could blend low level machine mechanics with high level language concepts. C has endured as a programming language because of its unique suitability for programming operating systems.

With recent algorithmic and hardware advances, C compilers have been able to catch up to Fortran compilers for efficient, computationally focused applications. This has made C the forerunner for current imperative programming.

C was also expanded to allow for object-oriented programming in the 1980's. C++ remains one of the most widely used object-oriented programming language today.

Compiling C

Like Fortran, and other high level languages, your program must be compiled into a binary before it can be executed. To do this, you have to invoke a C compiler on your source files. There are many C compilers on the market with different options and capabilities, but the one we will be using in this course is GNU C (gcc). To compile a file, simply type gcc <source file> on the command line. This will produce a binary executable named a.out. a.out can be executed like any other program (ie ./a.out).

Always producing an executable named a.out is not that convenient. You can change the name of the executable with the -o option. (gcc <source file> -o <executable name>)

➢gcc myProgram.c -o myProgram.exe

Compiling C

If you are writing a complex program, it is sometimes more convenient to write your code in chunks in different files. To compile your program in this case, you will need to first compile all the individual source files into object files. Object files are machine encoded binary files with a special table listing of all the variables in the file (which is used in linking). Once you have created object files of all your code, you need to link the files together to make a program (essentially mesh all the bits into a single executable file).

To compile an object file, use the -c option (gcc -c <source file>). You need to do this for every file in your program. The result of doing this is that a file will be created with the same name as your source file, but with .o as the extension. (ie gcc –c function.c would create function.o).

Compiling C

To link object files together, you just have to invoke gcc with the list of object files (gcc <object files>).

Much usable code has already been written into modules available as part of most compiler suites. If you are including modules in your code, you will need to instruct the compiler where to find the header files. Use the -I option for this when compiling source files (gcc -I<header path> <source file> ...) You can use the -c and -o options with -I. For more complicated compilations, it is customary to use makefiles, but this is outside the scope of this course.

```
➢gcc -I./myfiles -c myFunction1.c
➢gcc -I./myfiles -c myProgram.c
➢gcc myFunction1.o myProgram.o -o test.exe
```

Compiling C

Here are some examples:

```
➢gcc mySource.c
# this creates your program as a.out
➢gcc mySource.c -o myProgram
# this creates your program as myProgram
➢gcc -c mySource.c
# this creates an object file from mySource.c (mySource.o)
➢gcc -I/home/me -c mySource.c
# this creates an object file and includes any header files
# located in /home/me
➢gcc -c mySource1.c
➢gcc -c mySource2.c
➢gcc mySource1.o mySource2.o -o myProgram
# this compiles myProgram from the object files mySource1.o
# and mySource2.o
```

Setting Up a C Program: Main Function

Now that you understand how to compile a C program, we are going to start building up a C program piece by piece. First, we need the program framework – the main function.

All C programs require a main function. The C compiler recognizes the main function as forming the body of an executable binary (in other words, the framework of your program). The main function forms the outermost scope of the program, and so is the primary function you will declare in your program source file.

To declare the main function, use the line;

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

Following the line, all instructions between { and } form the body of your program. We will cover function syntax in more detail shortly and this line will make more sense.

Commenting a C Program

Commenting is always important for you and other people using your code!

In C, you create a comment using /* and */. Anything that appears between these character combinations is treated by the compiler as a comment and ignored. This is true, even if /* and */ occur on different lines. This can sometimes cause bugs in code where programmers accidentally comment out sections because they forget to close a comment with */.

Comments can appear on lines with or without code, before or after code on the same line.

In standard ANSI C, you can also comment out the remainder of a line using //. However, this form of commenting is not well supported by all C compilers (ie Portland Group).

ATSC 212 - C

Adding Modules: Include Statements

If you design a program with many functions defined in other source files you wrote, or you wish to use standard modules built in to C, you will need to include them in your program otherwise the compiler will be confused by any references you make to them.

To include additional source code, we include header files like this;

```
#include <header file>
```

If you are including standard C modules, then you should put < and > around the header file name. This tells the compiler to look in the usual places on the system for the header file. If you are using your own functions, put your header file name in double quotes ("), which tells the compiler to look in other locations you specify with the –I flag.

9

ATSC 212 - C

Adding Modules: Include Statements

Here are some common C module include statements you might find useful.

```
#include <math.h>      /* For C math functions */
#include <stdio.h>     /* For file handling functions */
#include <stdlib.h>    /* Has some conversion functions */
#include <limits.h>    /* Integer size parameters */
#include <float.h>     /* Float size parameters */
```

Include statements always appear outside of the main function and usually at the very beginning of source files. The functions for a given header are only available after the include call!

10

What We Have So Far

Here is what a sample C program would like like so far.

```
/* myCprogram.c */

#include <math.h>                 /* Math library */
#include <stdio.h>                /* File handling library */
#include <stdlib.h>               /* Standard library */
#include “myCfunctions.h”         /* My special functions */

int main (int argc, char* argv[])
{
    /* Where my code will go */
}
```

Adding Modules: Header Files

With the talk of include statements, you were probably wondering what a header file was. Because we do not want to have to write huge, single files containing an entire program and might like to put additional functions in their own files, we need a way to tell the compiler what these additional functions look like. We do this with header files.

Header files are files that contain function prototypes and constants. Header file names end with a .h instead of a .c to signify that they contain only prototypes and constants. **Header files are never compiled.** They are used by the compiler to build symbol tables. The –I flag we mentioned earlier is used to tell the compiler where to look for these files (if the files are not in the usual places on the system – ie /usr/local/include).

Adding Modules: Function Prototypes

A function declaration shows what the function returns, what variables it expects, and contains the code for the function. A function prototype shows only what the function returns and what variables it expects. Function prototypes are terminated with a semicolon. For example;

```
int power(int x, int y);
```

would be a function prototype for the function power.

You do not explicitly have to name the parameters in the function prototype (ie int power(int, int) is acceptable), however most people do for clarity.

Coding and Syntactical Standards

Now that we know how some basics for setting up C program files, it is time to cover some programming standards.

In C, blocks of code are encapsulated in curly brackets { }. The bodies of functions. and code blocks following conditionals and loops always occur in these brackets.

It is common practice to indent lines that occur in code blocks to help make different levels of scope stand out.

All regular statements are terminated by a semicolon. Forgetting to put a semicolon after a statement is a common bug. **Code blocks ({}), include statements, and comments are <u>not</u> terminated by a semicolon.**

Variables

Variables in C serve the same purpose as in other languages. However, there are several important aspects to C variables that can be different from languages we have seen.

Variables are strongly typed in C, so we must declare the variable and its type before we can use the variable. The available types in C are int, float, char, short, long, double, and void. int, short, and long are integer types. The difference is only in the range of values they will take and this dependent upon the system. float and double are real number types and differ in range of values like integer types. char is the character type, used to create strings. void is a null type that is used in sophisticated pointer code.

Variables

To declare a variable, use;

<type> <variable name>;

<type> is any of the types listed. <variable name> is the variable name. For example,

int air_temp;

Variables can also be assigned values when declared.

We reference and declare variables in C by their name (no need to use special symbols like $ or @ such as in PERL or bash).

Variable scope can sometimes give new programmers some trouble. A variable is only accessible within the code block where it was declared. So if you put a variable in a function outside your main program, you cannot access it in main.

ATSC 212 - C

Variables – Explicit Type Conversion

Because variables are strongly typed in C, you usually cannot use a variable of one type where another is expected, or set a variable to a value of another type. However, you can change how a variable or result is treated by casting it as another type. To do this, you preface the variable or result with (<type>). For example;

```
int a;
float b;
b = 2.5;
a = (int)b;
```

This method of casting is called explicit type conversion because you are telling the compiler to do the conversion. Some types of casting (such as float to char) will have unexpected results or are not possible.

17

ATSC 212 - C

Variables – Implicit Type Conversion

Type conversion can also be implicit between floats and integers. That is to say, you can assign an integer value to a float variable, and vice versa, without explicitly casting the value. If you assign a float value to an integer variable, the value will be truncated to the integer portion. This can lead to some roundoff errors in calculations if you are not careful. Otherwise, integers are treated as floats in calculations where a float is present. Results of calculations are always converted to the type of variable they are assigned to.

```
int a;
float b;
a = 5;
b = 3.2;
a = b;          /* a = 3 */
a = a * b;      /* a = 9 */
```

18

Scope

The term scope has been mentioned several times. Scope literally means 'where something can be seen'. This applies primarily to variables but can also apply to functions. When a variable is out of scope, the program or function does not know the variable exists. It is possible to have two variables with the same name in different scopes (although this often becomes confusing when reading the code).

Code blocks define scope. A variable defined within a code block can be seen everywhere in that code block but not outside it. For this reason, variables defined within functions cannot be seen by the main program and we will have to use some special tricks to get variables into and out of functions.

Scope is one area where programmers can trip themselves up. If you are running into problems with a variable, chances are it is out of scope.

Arrays

C also has arrays for data sets like the other languages we have seen. Arrays can be built up from any type, and can be multi-dimensional. To declare an array of a fixed size, use;

```
<type> <variable name>[size];
```

size tells how many elements the array can hold. Arrays declared in this fashion are static and cannot be made larger or smaller, so it is important to never attempt to add or read elements from an array greater than its size. For example, lets make a 10 element air temperature array.

```
float air_temp[10];
```

To reference an array element use <variable name>[index]. **In C, the indices go from 0 to (size - 1).**

Arrays

Any element of an array can be changed or set, as if they were a simple variable, by referencing that index. Here is an example:

```
int a[2];
a[0] = 1;
a[1] = a[0] + 2;
```

Multi-dimensional arrays can also be used in C. To declare them, add more [size] parts to the declaration. For example:

```
int a[12][5][7];
```

creates a three dimensional array of integers (12 x 5 x 7).

Arrays

One of the biggest differences between Fortran and C is the manner in which arrays are handled in memory. Fortran stores arrays in column-major format while C stores them in row-major format (usually). You do not really have to worry about this for coding pure C or Fortran programs, nor does it affect performance. However, as we will see later, this presents a large problem for interfacing C to/from Fortran code.

A final note, strings in C are arrays of characters. The strings.h code adds additional functionality and a type for strings, but most C programmers simply handle strings as character arrays.

ATSC 212 - C

Operators

Now that we have variables, it is time to do something with them. Operators are like built in functions that will perform an action on variables. There are two types of operators in C, mathematical and logical. All operators can be used on integers, floats, and characters, and to some extent, pointers.

Mathematical operators include =, -, --, +, ++, *, /, %, -=, +=, *=, and /=. The % operator is the modulus. -=, +=, *= and /= apply the mathematical operator to the value on the left with whatever is on the right (ie a *= b; is the same as a = a * b;). Care must be taken when using anything but = with characters and pointers. Results are system dependant. These operators work with regular variables or array elements. The ++/-- operators increment/decrement a variable by 1.

Logical operators include <, <=, ==, !=, >=, >. There are no logical operators for strings, however, there are functions for comparing strings as we will see.

23

ATSC 212 - C

Conditionals

It is all about choice. Just as with other languages, sometimes we would like to choose what actions we take. C has one conditional – the if statement:

```
if (<condition>) { …code block… }
else if (<condition>) { …code block… }
else { …code block… }
```

The else if and else portions are optional. C is different from some other languages in that it does not have a true boolean type. **Any condition that evaluates to 0 is false, all others are true.** The logical and mathematical operators can be used to form conditions. This includes assignment (=) and equality (==). A big pitfall of programmers is accidentally using assignment where they mean to compare two things. The condition a == b checks if a and b are the same. The condition a = b sets the value of a to the value of b and returns true.

24

Conditionals

Here is an example:

```
if (air_temp < 0)
{
  /* some code to handle negative temperatures */
}
else if (air_temp == 0)
{
  /* handle the special case of 0C */
}
else
{
  /* air_temp must be greater than 0 in this case */
}
```

Loops

There are two types of loops in C, while and for. While functions loop until a condition is false (or the loop is broken out of). The for loop works similar to the for loop in Fortran, iterating a number of times. The syntax for the loops are:

```
while (<condition>) { …code block… }

for (<variable> = <start>; <condition>; <alteration>)
{ …code block… }
```

To make a for loop, you need a variable that you can set to a starting value, <start>. Typically, the for loop iterates until some condition dependant upon the variable is met. Each iteration of the loop makes an alteration, usually to the variable.

Loops

Here's an example of a for loop:

```
int i;
for (i = 0; i < 12; i++)
```

This sets i to 0. Each loop adds 1 to i. For will continue looping until i = 12 at which point the condition will be false. By changing the condition, starting point, and alteration, for loops can count up or down by a variety of values from any starting value.

You can do the same thing in a while loop like this:

```
int i = 0;
while (i < 12) { … i++; }
```

Loops

There are two additional keywords in C that can affect loop behaviour, break and continue.

If you want to immediately leave a loop, you can add the line:

```
break;
```

This will cause your program to exit the loop and continue running the program just after the loop code. This can be used to add extra conditions to how a loop terminates other than those in the for or while declaration.

continue; allows you to skip to the next iteration of a loop without running the rest of the code within the loop. This can be useful if there is a particular condition in which you do not want to run that code but want to continue looping.

Loops

Here are some examples;

```
i = 13;
while (i >= 12) /* Keep looping as long as i >= 12 */
{
  ...                    /* Some instructions in the loop, one
                              of which sets i to a new value */
  if (i < 0) { break; }  /* Check if i < 0, if so, quit looping */
}

for (i = 0; i < 5; i++)          /* Loop from i = 0 to i = 4 */
{
  if (i == 3) { continue; }      /* If i = 3, skip to next loop */
}
```

Functions

Functions are like Fortran subroutines. They allow us to break larger programs into smaller pieces, and to encapsulate re-useable code into discrete sections. You have already seen one function that occurs in every C program, the main function.

To declare a function, use:

```
<type> <function>(<list of parameters>) { …code block… }
```

Functions can only return one value in C. <type> determines the type of the value returned. <function> is the name of the function. The list of parameters is a comma delimited list of <type> <variable> pairs that show everything the function expects to be called with.

ATSC 212 - C

Functions

For example;

```
int power(int x, int y)
```

declares the function power to return an integer value, and take two integer parameters x and y.

After the function declaration, the code block describes the function. The parameters listed in the function declaration are accessible variables within the code block. So in the example above, x and y are already declared variables that can be used in the function code block.

31

ATSC 212 - C

Functions

To call a function in C, you put a statement with the function name, and values or variables of the appropriate type for each parameter. If you do not supply the appropriate values or variables for the parameters, the code will not compile. For example, we would use the power function like this.

```
int a, b, c;
a = 5;
b = 4;
c = power(a, b);
```

32

Functions

In C, it is possible to have several functions with the same name but different parameter listings. This is referred to as function overloading. When you compile, the compiler will look for a function declaration that matches the list of parameters you supplied, and use that function. You cannot declare functions of the same name with the same parameter listing.

As mentioned, functions return one value. They way they do this is with a return statement inside the code block.

```
return <value>;
```

Functions

<value> must be of the same type as the function was declared as, otherwise implicit type casting may occur. You can have multiple return statements within a code block. The first time a return statement is executed, the function will exit.

But what if we want to return more than one value? The usual way around this problem in any language is to change the variables that are passed as parameters to the function. The problem with C functions is that they pass-by-value. What this means is that when we make a function call, the parameters we pass to the function are copied into the variables in the function declaration. The original variables passed in are not changed by the function.

Functions

To get around this, we need to pass-by-reference, which is what Fortran does. Passing by reference means that instead of passing actual values, we pass locations of values (called addresses). The trick to accomplishing this in C is pointers, which we will discuss shortly.

Finally, you need to be careful about scope when composing functions. You cannot call a function in a program unless the declaration or prototype appears before the function call. This is why we use header files and put #include statements at the top of source files. This allows function prototypes to appear at the beginning of a source file before we attempt to use them. Failure to declare a function or prototype before calling a function will cause the compiler to fail.

Functions

Here is an example function;

```
int power(int x, int y)
{
  /* This function returns the value of x to the power of y, y >= 0 */
  /* The variables x and y are set by the parameters passed in on
       the function call */
  int counter;       /* A loop counter.  Only valid within this function. */
  int result = 1;    /* The result of x to the power of y */

  for (counter = 1; counter <= y; counter++)
  {
    result *= x;
  }
  return result;
}
```

String Functions

Because strings are arrays of characters, we cannot simply assign a quoted string to a character array using standard assignment or compare two strings using == or eq.

```
char myname[20];
myname = “George”;  /* THIS DOES NOT WORK */
if (myname == “George”) /* THIS ALSO DOES NOT WORK */
```

However, there are functions built into stdio.h and string.h that will allow us to do these things.

String Functions

To put a string into a character array, we can use the function sprintf.

```
sprintf(<array>, “<string>”, <variables>);
```

<array> is our character array. It should be at least one character larger than the number of characters in the string we are going to store in it. <string> is a specially formatted string. It will contain all the text we want to appear in our string, plus special format characters for any variable values we want in the string. <variables> is just a comma delimited list of variables appearing in <string> in the order in which they appear.

ATSC 212 - C

String Functions

In C, we cannot simply put variables into the format string and have them dereferenced as their values. To tell the function that a particular value goes in the string, we use the following codes:

%f	this tells sprintf to put a float value here
%d	this tells sprintf to put an integer value here
%s	this tells sprintf to put a string (character array) here

%f can also take numbers specifying how to print the float. %x.yf would indicate that the float should be written as having x characters (including the decimal) and should have y digits after the decimal. So %6.4f would write out 1.5 as 1.5000.

39

ATSC 212 - C

String Functions

%xd can also be used to specify the number of characters to be printed for a given integer. Here's an example.

```
int a = 5;
float b = 10.5;
char output[30];
char more[50];

sprintf(output, "A is %d and B is %4.1f.\n", a, b);
sprintf(more, "%s And A/B is %f.\n", output, (a/b));
```

40

ATSC 212 - C

String Functions

To compare two strings, we use the strcmp function. The function call looks like:

strcmp(<string1>, <string2>)

This function sometimes confuses new programmers because it returns 0 if the strings match and a non-zero integer if the strings are not equal (the number you get back is based on a character by character comparison and is case sensitive). So if we wanted to check if two strings match, we would use if (strcmp(<string1>, <string2>) == 0). For example,

```
if (strcmp(var, “Air Temperature”) == 0) { ... }
```

would do something if var was the string “Air Temperature”.

41

ATSC 212 - C

String Functions

Although there are many useful string functions, we only have time for one more, printf. printf allows you to print a string to the screen.

printf(<format string>, <variables>);

The <format string> and <variables> are handled just as with sprintf. For example:

```
printf(“The wind speed is: %7.2f\n”, wind_spd);
```

would print a line like The wind speed is: 25.30 to the screen.

42

ATSC 212 - C

File Handling

To use files in C, we need to make sure we include the stdio.h. This contains the function and type definitions.

To work with a file we need a filehandle. In C, filehandles are regular variables with the type FILE* (this is a special type defined in stdio.h). You declare filehandles with other variables at the beginning of a code block.

To open a file, we use the fopen function:

```
<filehandle> = fopen(<filename>, <mode>);
```

For example we could open input.txt for reading,

```
FILE* myfile;
myfile = fopen("input.txt", "r");
```

43

ATSC 212 - C

File Handling

<filehandle> is the variable we declared to hold the filehandle. <filename> is the string for the filename. This should be the full path to the file. If it is not, then the program will assume the file is located within the directory the program is run from. <mode> is a string that tells how the file is opened. The different modes are:

"r"	opens the file in read-only mode
"w"	opens the file in write-only mode, clears the contents
"a"	opens the file in write-only mode, appends to the end
"r+"	opens the file for reading and writing

Normally, fopen opens a file for reading/writing in ASCII format. If we want to treat the file as a binary, we add b to the mode. (ie "rb", "wb")

44

File Handling

Once we are done using a file, we need to close it. To close a file, use the fclose function.

```
fclose(<filehandle>);
```

fclose does return 0 if the file closed successfully and a non-zero integer if it failed to close the file properly. However, programmers rarely check the status of closing files because it is usually difficult or impossible to correct a failed file closure.

Continuing the previous example, we can close myfile like this,

```
fclose(myfile);
```

File Handling

To write to a file in ASCII format, we use fprintf. This is almost exactly like sprintf and printf.

```
fprintf(<filehandle>, <format string>, <variables>);
```

```
FILE* myfile;
int air_temp = 25;

myfile = fopen("output.txt", "w");

fprintf(myfile, "This is my output file!\n");
fprintf(myfile, "Air Temperature is %5d\n", air_temp);

fclose(myfile);
```

ATSC 212 - C

File Handling

Reading from a file is somewhat more difficult. There are three functions commonly used for reading in ASCII formatted data into strings; fscanf, fgets, and fgetc.

fscanf is similar to fprintf. It takes a <format string> that defines what we expect to read in, and then puts the data it finds into the <variables> listed.

```
fscanf(<filehandle>, <format string>, <variables>);
```

For example, suppose we want to read air temperature, relative humidity, and wind speed off a line (assuming temperature and humidity are integers and wind speed is a float),

```
fscanf(myfile, "%d %d %f", &air_temp, &rel_hum, &wind_spd);
```

47

ATSC 212 - C

File Handling

The main difference between fprintf and fscanf is that each variable must be a pointer. We will discuss these later.

fgets will read in a string from a file of up to n characters in length. fgets stops reading if it reads a newline. Newlines are included in the string if read.

```
fgets(<array>, n, <filehandle>);
```

<array> is the character array we have declared to hold the string. <array> must be at least n+1 characters in size. If the line from <filehandle> is larger than n, then only n characters are read from the filehandle.

48

ATSC 212 - C

File Handling

For example, suppose the filehandle myfile points to a file containing (with newlines '\n' explicitly indicated);

These are the contents of my file.\n
There are no other contents.\n

Now if we had a character array buffer[30], then

```
fgets(buffer, 29, myfile);
// buffer would be "These are the contents of my "

fgets(buffer, 29, myfile);
// buffer would be "file.\n"

fgets(buffer, 29, myfile);
// buffer would be "There are no other contents.\n"
```

49

ATSC 212 - C

File Handling

fgetc reads the next character from a file and returns it as an unsigned integer (which can be recast directly or implicitly as a character).

```
<character> = fgetc(<filehandle>);
```

This function can be extremely tedious to use for files that have single records on each line as it can take many calls to pull the entire line out one character at a time. It is usually used to implement specialized token retrieval.

50

Getting Under the Hood: Pointers

So far, C has probably looked a lot like other programming languages you have seen, with slight differences in syntax or particular functions to perform things like file handling. Now we reach the most significant difference between C and many other high level programming languages, pointers.

Pointers expose some of the underlying machine mechanics to us to allow us to perform some very powerful, and potentially disastrous, code tricks.

A pointer is an address in memory. You can think of memory in a computer as being a city. Each house holds a particular amount of data (bytes). The data could be any type.

Getting Under the Hood: Pointers

Each house has an address, just as it would in a real city. When the computer is performing a calculation, it needs to go to the relevant houses and get the data contained there. When the calculation is done, it goes to the house that stores the result and puts it there.

All of this has been abstracted behind variables in other programming languages. This is equally true in C when you are using a variable of a particular type like integer. Essentially, when you declare a variable in C, you are saying to the computer to associate a house with that variable that will hold a particular type of data. You do not care which house, and you never worry about that. The computer handles all that for you.

Getting Under the Hood: Pointers

Okay, I get it...pointers are addresses in memory. So an integer pointer would be a location in memory that holds an integer. That is precisely correct!

But why would I want to know the address rather than the value? What can pointers do for me?

Pointers can be used for many tricks in memory but the two important uses we will focus on are dynamic memory allocation, and multi-value functions (also called procedures).

Declaring Pointers and *

Before we get into how to use pointers, lets look at how to declare them. To declare a variable as a pointer, we use:

```
<type> * <name>;
```

The * acts as a dereferencing keyword in this context. It tells C that our variable is a pointer. * is also used to get at the value of a pointer when used outside of type declaration (remember, pointers are addresses).

For example;

```
int * x;          /* x is a pointer to an integer */
char * me:        /* me is a pointer to a character */
float * result;   /* result is a pointer to a float */
```

ATSC 212 - C

Declaring Pointers

Pointers variables can be made to any type. For added clarity, programmers usually put the * symbol adjacent to either the type or the variable name when declaring the variable. Either format is acceptable, although syntactically it is not necessary.

```
int* x;          /* These declarations all create an
int * x;            integer pointer called x */
int *x;
```

The main thing to keep in mind is that the value a pointer variable holds is an address. If you change the value of a pointer variable, you are changing the address it stores (or as we say, points to). Pointer variables can be used to point to different areas of memory, but to point to the address of an existing piece of memory, we have to know what the address is. We use the reference keyword & to get the address of a variable. Lets look at an example.

55

ATSC 212 - C

Pointers

Consider the following code:

```
int x;                      /* x is an integer */
int* y;                     /* y is an integer pointer */
x = 5;                      /* x has a value of 5 */
y = &x;                     /* y now points to x */
printf("%d\n", *y);         /* we print the value of the
                               the address y points to by
                               using the '*' before y, this
                               prints '5' */
x = 7;                      /* x now has a value of 7 */
printf("%d\n", *y);         /* y points to x, so it prints 7 */
```

56

Dynamic Memory Allocation

Thusfar, you have seen that if you want to create an array, you have to declare it as a certain size. This is not very flexible. If you do not know how big the array might be, you would have to guess in your code and make the array at least as big as you imagine it could be. This can make your program take a lot of memory.

Dynamic memory allocation allows you to ask the computer to give you memory as you need it. A pointer can then be used to show you where the memory you asked for is and act as an array.

The function that gets memory for you is malloc (short for memory allocation). The function that frees memory you are done using is free.

Dynamic Memory Allocation

Here is how you use them:

```
<pointer> = (<type cast>*)malloc(sizeof(<type>) * <size>);

free(<pointer>);
```

<pointer> is your variable that you want to hold the location of memory.

<type cast> is the type you want to treat the memory as. For example, if we are creating an array of integers, we want to type cast as int. <type> is the same type as <type cast>. <size> is the size of the array you wish to create.

Dynamic Memory Allocation

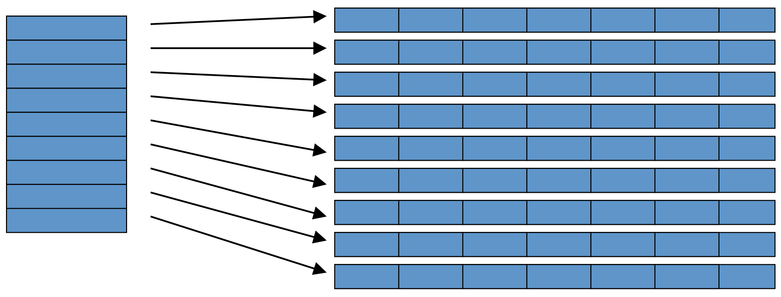
The sizeof function is a special function that tells you how many bytes there are for a given type. The value it returns is system dependant, so it allows you to write flexible code that will always get as much memory as you need for an array of the type you want. Here is an example,

```
int* air_temp;
int num_values;
...
fscanf(myfile, "%d", &num_values);
air_temp = (int*)malloc(sizeof(int) * num_values);
...
free(air_temp);
```

Dynamic Memory Allocation

It is possible to dynamically allocate multi-dimensional arrays. To do this, we need to use pointers that point to pointers.

For example, to get a two dimensional array, we would need to allocate a single dimensional array (like a column) of pointers. Each pointer in this array would point to a row. Like this:



ATSC 212 - C

Dynamic Memory Allocation

The way we do this in code is that we dereference each dimension. So two dimensional arrays would have pointers like `int** x` and `float** y`. Three dimensional arrays would have pointers like `int*** x`.

To allocate memory for these arrays, we have to do them one dimension at a time, looping over each dimension but the outermost. Say we have a two dimesional array that is x by y.

```
int** myArray;
int x, y, i;

myArray = (int**)malloc(sizeof(int*) * x);
for (i = 0; i < y; i++)
{
   myArray[i] = (int*)malloc(sizeof(int) * y);
}
```

61

ATSC 212 - C

Dynamic Memory Allocation

Freeing the array works in reverse. First we need to free each row, then free the overall array.

```
for (i = 0; i < y; i++)
{
   free(myArray[i]);
}
free(myArray);
```

Higher dimensional arrays simply add more loops and more dereferencing.

62

A Simple Example

Bringing it all together:

```
/* Declare the pointer */
int* x;

/* Get space for an array of 100 integers */
x = (int*)malloc(sizeof(int) * 100);

/* Set some values */
x[0] = 1;          /* [0] is the first element */
x[99] = 100;      /* [size-1] is the last element */

/* Prints "First 1, Last 100" */
printf("First: %d, Last: %d\n", x[0], x[99]);

/* Free the array */
free(x);
```

Pass-by-Reference

The other important area is multi-value functions. Remember that functions in C only return one value and parameters passed to functions are passed-by-value (copied). The way to return multiple values is to change the parameters passed to functions. The way to do this is to pass-by-reference. We will use pointers as parameters.

If we use pointers (addresses) as parameters, then what is copied in to the function is the addresses of values we would like to change. The addresses are immutable, but we can change whatever is at those addresses (circumventing the pass-by-value paradigm of C). For example, we could have the power function return the result in a variable called result.

```
int power (int x, int y, int* result);
```

ATSC 212 - C

A Simple Example

Let's look at a multi-value function example. Suppose we alter the power function so that it returns 0 if successful, and 1 if it fails. The value of x to the power of y is returned in the parameter result.

65

ATSC 212 - C

Multi-Valued Power Function

```
int power(int x, int y, int* result)
{
  /* Calculates x^ y (y >= 0) and returns the result in the parameter result. */
  int counter;          /* A counter for looping */

  /* Check if y < 0 since we won't calculate a result if it is.  If y < 0, return 1. */
  if (y < 0) { return 1; }

  /* y >= 0.  Set the result to 1 to start. */
  *result = 1;  /* This has changed the value at the address pointed to by result.
                   Even the main program will see this change. */

  /* Multiply result by x, y times. */
  for (counter = 1; counter <= y; counter++)
  {
    *result *= x;
  }

  /* We are finished, and successful, so return 0 */
  return 0;
}
```

66

C vs Fortran

You have seen the key elements and syntax that make up C. How does it compare to Fortran?

Both languages have similar looping and conditional structures. Both languages have types for integers, floats, and characters. However, C is strongly typed while Fortran can be context based.

Both languages have arrays for data. Fortran 95 also has dynamic memory allocation like C.

Both languages have built-in functions and libraries that perform such things as mathematical calculations or file handling.

C vs Fortran

Until probably about ten years ago, Fortran was still the premier language for large-scale calculation based computation. However, with improvements in compilers, many high level languages, including C, now boast the calculation throughput of Fortran.

Thus the primary difference between C and Fortran is pointers. C has them, while Fortran does not fully support them. For scientific programming purposes, this actually is not important. Fortran 95 supports dynamic memory allocation and multi-valued subroutines while abstracting away the underlying pointer mechanics which makes it an equal to C for most programmers.

Fortran was the primary language of calculation based programming and so there is considerable old code still being used today written in Fortran. As a result, much of the new code being written to interface with this older code is also in Fortran. C is one of the most widely used programming languages today, but not within these environments.

Meshing C with Fortran

Most languages gradually change over time to improve performance or add new features. Fortran did not have dynamic memory allocation until Fortran90 (a version of Fortran rarely used). Fortran95, originally released in 1995, has added much of the functionality of C missing from previous versions of Fortran. However, it was not until 2003 that non-proprietary Fortran95 compilers became available.

Prior to that time, if programmers or scientists wanted to add dynamic memory allocation to their code, or other functionality through pointers, they would have to write C code and then link that code to existing Fortran code. As a result, there now exists a lot of scientific code written in both C and Fortran.

Meshing C with Fortran

Normally, this is done by compiling all code into object files and then using a special program called a linker to put the files together. (Linkers are usually part of compilers).

To make this work, we have to understand a bit about how Fortran works under the hood, because the Fortran compiler will do things when compiling that the C compiler will not. As such, we are going to have to do things in C to accommodate Fortran.

First, recall that Fortran functions always pass-by-reference. This means that if we write a function in C that is called in Fortran, all of its parameters must be pointers.

Meshing C with Fortran

If we want to call a Fortran function in C, we must pass in pointers. As far as Fortran is concerned, parameters and returned values are just standard types.

The next difference is how the compilers build their symbol tables. When Fortran compiles functions into machine code, it converts the name to lowercase and adds an underscore to the end of function names. (ie the function POWER becomes power_) C compilers do not do this. To accommodate Fortran, we need to make function names lowercase and add an underscore to the end of functions written in C to be called by Fortran. We also need to make Fortran function calls in C in lowercase with an appended underscore.

Meshing C with Fortran

I have been mixing the concept of Fortran subroutines and functions together here because C regards both ideas as functions. As far as C is concerned, Fortran functions are regular C functions (with the noted differences above). Fortran subroutines are treated by C as functions with a return value of void (void is a special type with no value).

The final difference is how C and Fortran treat arrays. Fortran starts array indexing at 1 and C starts array indexing at 0. For example, the first element of an array data in Fortran is data[1] while in C is data[0]. Coincidentally, the last element of the array in Fortran is the size of the array, while it is the size-1 in C. This is one of the single biggest causes of confusion and errors in cross coding between the languages.

ATSC 212 - C

Meshing C with Fortran

One dimensional arrays in C and Fortran look pretty much the same other than the indexing issue. However, multi-dimensional arrays look different in memory. C is row-major meaning that it will store arrays in memory row by row. Fortran is column-major. Here is an example to demonstrate the problem.

data =

4	18	3
12	5	1

In memory, C writes data out as 4 18 3 12 5 1, while Fortran writes data out as 4 12 18 5 3 1.

73

ATSC 212 - C

Meshing C with Fortran

In general, C writes arrays out by last index first while Fortran writes arrays out by first index first.

In practice, this means if you want to pass multi-dimensional arrays between C and Fortran, you need to write your C code with the indices in reverse. So if we wanted to represent the data array from the previous slide in C, we would make it 3x2 instead of 2x3 and treat the first index as the column rather than the row.

In Fortran	In C
data(1,1)	data[0][0]
data(1,2)	data[1][0]
data(1,3)	data[2][0]

In Fortran	In C
data(2,1)	data[0][1]
data(2,2)	data[1][1]
data(2,3)	data[2][1]

74

Although you probably feel overwhelmed by what we have covered, we have only just scratched the surface of C. I highly recommend Kernighan and Ritchie's book *The C Programming Language* if you intend to do extensive C programming. You can also find more detailed information and examples of meshing C and Fortran at http://www.aei.mpg.de/~jthorn/c2f.html.