CS50 notes from Book

Typical errors reported during this phase of compilation might be due to an expression that has unbalanced parentheses (**syntactic error**), or due to the use of a variable that is not "defined" (**semantic error**).

The C language allows data types other than just integers to be stored in variables as well, <u>provided the proper **declaration** for the variable is **made before** it is used in the program</u>. Variables

Each data type requires a certain amount of system resources:

- Bool = 1 byte
- int = 4 bytes
- long = 8 bytes
- float = 4 bytes
- double = 8 bytes
- char = 1 byte

The rules for forming variable names are quite simple: They must begin with a letter or underscore (_) and can be followed by any combination of letters (upper- or lowercase), underscores, or the digits 0–9.The following is a list of valid variable names.

sum
pieceFlag
i
J5x7
Number_of_moves
_sysflagstring = ? bytes

On the other hand, the following variable names are not valid for the stated reasons

sum$value	$ is not a valid character.
piece flag	Embedded spaces are not permitted.
3Spencer	Variable names cannot start with a number.
Int	int is a reserved word.

Note: upper- and lowercase letters are distinct in C

variable **int** can be used to contain integral values only—that is, values that do not contain decimal places.
variable **float** can be used for storing floating-point numbers (values containing decimal places).
variable **double** is the same as type float, only with roughly twice the precision.
The **char** data type can be used to store a single character, such as the letter a, the digit character 6, or a semicolon.
the **Bool** data type can be used to store just the values 0 or 1 indicating an on/off, yes/no, or true/false

In C, any number, single character, or character string is known as a **constant**.
Expressions consisting entirely of constant values are called **constant expressions**.

e.g.
128 + 7 – 17 is a constant expression because each of the terms of the expression is a constant value.

But if **i** were declared to be an integer variable, the expression

128 + 7 – i

would not represent a constant expression.

an integer constant consists of a sequence of one or more digits.

- A minus sign preceding the sequence indicates that the value is negative.
- No embedded spaces are permitted between the digits, and values larger than 999 cannot be expressed using commas.
  (So, the value 12,000 is not a valid integer constant and must be written as 12000.)

- If the first digit of the integer value is a 0, the integer is taken as expressed in octal notation—that is, in base 8.
  - In that case, the remaining digits of the value must be valid base-8 digits and, therefore, must be 0–7.
  - So, to express the value 50 in base 8 in C, which is equivalent to the value 40 in decimal, the notation 050 is used. Similarly, the octal constant 0177 represents the decimal value 127 (1 × 64 + 7 × 8 + 7).
  - An integer value can be displayed at the terminal in octal notation by using the format characters %o in the format string of a printf statement.
  - In such a case, the value is displayed in octal without a leading zero .The format characters <u>%#o</u> do cause a leading zero to be displayed before an octal value. If an integer constant is preceded by a
- If an integer constant is preceded by a zero and the letter x (either lowercase or uppercase), the value is taken as being expressed in hexadecimal (base 16) notation.
  - Immediately following the letter x are the digits of the hexadecimal value, which can be composed of the digits 0–9 and the letters a–f (or A–F).
  - The format characters %x display a value in hexadecimal format without the leading 0x, and using lowercase letters a–f for hexadecimal digits.
  - To display the value with the leading 0x, you use the format characters %#x, as in the following:
    - printf ("Color is %#x\n", rgbColor);
  - An uppercase x, as in %X, or %#X can be used to display the leading x and the hexadecimal digits that follow using uppercase letters.

Storage Sizes and Ranges

Every value, whether it's a character, integer, or floating-point number, has a range of values associated with it. This range has to do with the amount of storage that is allocated to store a particular type of data.

- In general, that amount typically depends on the computer you're running. 32 or 64 bits
- You should **never** write programs that **make any assumptions** about **the size of your data types**
- You are guaranteed that a minimum amount of storage will be set aside for each basic data type.
- it's guaranteed that an integer value will be stored in a minimum of 32 bits (4 bytes) of storage.

Floats

- A variable declared to be of type float can be used for storing **values containing decimal places.**
- You can omit digits before the decimal point or digits after the decimal point, but obviously you can't omit both.
- The values 3., 125.8, and –.0001 are all valid examples of floating-point constants.
- To display a floating-point value at the terminal, the printf conversion characters **%f** are used.
- Floating-point constants can also be expressed in scientific notation
  - The value 1.7e4 is a floating-point value expressed in this notation and represents the value 1.7 × 10–4.
  - The value before the letter e is known as the **mantissa**,
  - Whereas the value that follows is called the **exponent.** The E can be upper or Lower
  - This exponent, which can be preceded by an optional plus or minus sign, represents the power of 10 by which the mantissa is to be multiplied.
  - E.G.
    - The constant 2.25e–3, the 2.25 is the value of the mantissa and –3 is the value of the exponent.
    - This constant represents the value 2.25 × 10–3, or 0.00225.
- To display a value in scientific notation, the format characters **%e** should be specified in the printf format string
- The printf format characters **%g** can be used to let printf decide whether to display the floating-point value in normal floating-point notation or in scientific notation.
  - This decision is based on the value of the exponent: If it's less than –4 or greater than 5, %e (scientific notation) format is used; otherwise, %f format is used.
- Use the **%g** format characters for displaying floating-point numbers—it produces the most aesthetically pleasing output
- A hexadecimal floating constant consists of a leading 0x or 0X, followed by one or more decimal or hexadecimal digits, followed by a p or P, followed by an optionally signed binary exponent.
  - For example, 0x0.3p10 represents the value 3/16 × 210 = 192.
- Unless told otherwise, **printf always displays a float or double** <u>value to **six decimal places rounded**.</u>

Double

Type double is very similar to type float, but it is used whenever the range provided by a float variable is not sufficient.

- Variables declared to be of type double can store roughly twice as many significant digits as can a variable of type float.
- Most computers represent double values using 64 bits
- Unless told otherwise, all floating-point constants are taken as double values by the C compiler.
- To explicitly express a float constant, append either an f or F to the end of the number, as follows:
  - 12.5f
- To display a double value, the format characters **%f, %e, or %g,** which are the same format characters used to display a float value, can be used.

Char

- A **char** variable can be used to store a single character.
- A character constant is formed by enclosing the character within a pair of single quotation marks.
    - So 'a', ';', and '0' are all valid examples of character constants.
- Do not confuse a **character constant, which is a single character enclosed in single quotes**, with a **character string**, which is any number of characters enclosed in double quotes.
- The character constant '\n'—the newline character—is a valid character constant even though it seems to contradict the rule cited previously.
    - In other words, the C compiler treats the character '\n' as a single character, even though it is actually formed by two characters.
- The format characters **%c** can be used in a printf call to display the value of a char variable at the terminal.

Bool

- A **Bool** variable is defined in the language to be large enough to store just the values 0 and 1.
- By convention, **0** is used to indicate a **false** value, and **1** indicates a **true** value.
- When assigning a value to a _Bool variable, a value of 0 is stored as 0 inside the variable, whereas **any nonzero value is stored as 1.**
- To make it easier to work with _Bool variables in your program, the standard header file defines the values bool, true, and false.
    - As a binary operator, it is used for subtracting two values; as a unary operator, it is used to negate a value.
- The last printf shows that a _Bool variable can have its value displayed using the integer format characters %i

The most common of these ``character escapes'' are:

\n	**a ``newline'' character**
\b	**a backspace**
\r	**a carriage return (without a line feed)**
\'	**a single quote (e.g. in a character constant)**
\"	**a double quote (e.g. in a string constant)**
\\	**a single backslash**

Arithmetic Operators

Unary operator

A unary operator is one that operates on a single value, as opposed to a binary operator, which operates on two values.

The **minus sign actually has a dual role**:

The unary minus operator has higher precedence than all other arithmetic operators, except for the unary plus operator (+), which has the same precedence.

The Modulus Operator

- symbolized by the percent sign (%).
- The modulus operator can only be applied to integers.
- As you know, printf uses the character that immediately follows the percent sign to determine how to print the next argument.
- However, if it is another percent sign that follows, the printf routine takes this as an indication that you **really intend to display a percent sign (%%)** and inserts one at the appropriate place in the program's output.
- the function of the modulus operator % is to give the remainder of the first value divided by the second value.
- any operations between two integer values in C are performed with integer arithmetic.
- Therefore, any remainder resulting from the division of two integer values is simply discarded.

exponentiation

languages usually have an exponentiation operator (typically ^ or **),

but C doesn't have an exponentiation operator

To square or cube a number, just multiply it by itself.

Precedence

- Multiplication, division, and modulus all have higher precedence than addition and subtraction.
- All of these operators ``group'' from left to right, which means that when two or more of them have the same precedence and participate next to each other in an expression,
- the evaluation conceptually proceeds from left to right.
- For example, 1 - 2 - 3 is equivalent to (1 - 2) - 3 and gives -4, not +2.
- Whenever the default precedence or associativity doesn't give you the grouping you want, you can always use explicit parentheses.

Assignment Operators

i = i + 1   == i++

Function Calls

The arguments to a function can be arbitrary expressions. Therefore, you don't have to say things like

```
int sum = a + b + c;

printf("sum = %d\n", sum);
```

if you don't want to; you can instead collapse it to

```
printf("sum = %d\n", a + b + c);
```

Converting Between Integers and Floats

To effectively develop C programs, you must understand the rules used for the implicit conversion of floating-point and integer values in C.

Type	**Constant Examples**	**printf chars**
`char`	`'a', '\n'`	`%c`
`_Bool`	`0, 1`	`%i, %u`
`short int`	—	`%hi, %hx, %ho`
`unsigned short int`	—	`%hu, %hx, %ho`
`int`	`12, -97, 0xFFE0, 0177`	`%i, %x, %o`
`unsigned int`	`12u, 100U, 0XFFu`	`%u, %x, %o`
`long int`	`12L, -2001, 0xffffL`	`%li, %lx, %lo`
`unsigned long int`	`12UL, 100ul, 0xffeeUL`	`%lu, %lx, %lo`
`long long int`	`0xe5e5e5e5LL, 500ll`	`%lli, %llx, &llo`
`unsigned long long int`	`12ull, 0xffeeULL`	`%llu, %llx, %llo`
`float`	`12.34f, 3.1e-5f, 0x1.5p10, 0x1P-1`	`%f, %e, %g, %a`
`double`	`12.34, 3.1e-5, 0x.1p3`	`%f, %e, %g, %a`
`long double`	`12.341, 3.1e-51`	`%Lf, $Le, %Lg`

Logical Operators

A	B	A && B	A ll B	!A
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

https://medium.com/@JKDMarks/logic-truthiness-logical-operators-and-readability-8b1c13ce75e9

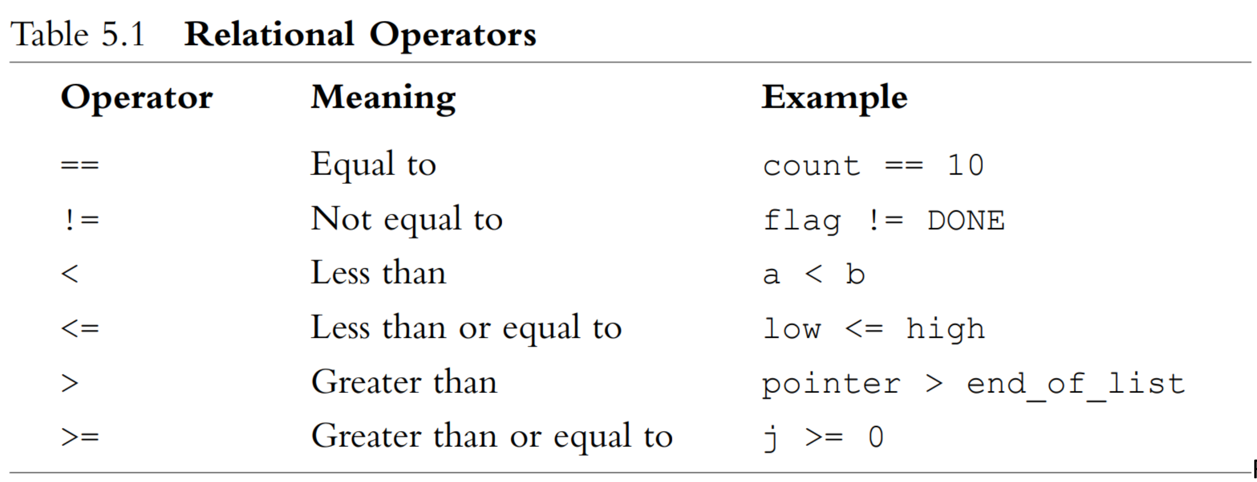


Table 5.1 **Relational Operators**

Operator	Meaning	Example
`==`	Equal to	`count == 10`
`!=`	Not equal to	`flag != DONE`
`<`	Less than	`a < b`
`<=`	Less than or equal to	`low <= high`
`>`	Greater than	`pointer > end_of_list`
`>=`	Greater than or equal to	`j >= 0`

For Loops

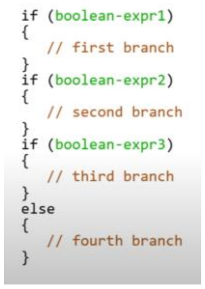
Relational Operators
Less than (x < y)
Less than of equal to (x <= y)
Greater than (x > y)
Greater than or equal to (x >= y)
Equality (x == y)
Inequality (x != y)

Conditionals

Conditional expressions allow your programs to make decisions and take different forks in the road depending on the values of the variables or user input
C provides a few different ways to implement conditional expressions (also known as branches) in your programs.

```
if (Boolean-expression)
{
        If the Boolean-expresion evaluates to true, all lines code in this{} execute in order (top to bottom)
}
else
{
        If the Boolean-expresion evaluates to false, all lines code in this {} execute in order (top to bottom)
}
```

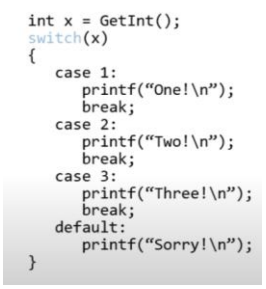
Its also possible to create a chain on non-mutually exclusive branches
In this example, only the third and forth branches are mutually exclusive



```
if (boolean-expr1)
{
   // first branch
}
if (boolean-expr2)
{
   // second branch
}
if (boolean-expr3)
{
   // third branch
}
else
{
   // fourth branch
}
```

The fourth branch starting with else only binds to the if statement directly above it, it does not bind to a chain of all 3 previous if statements.

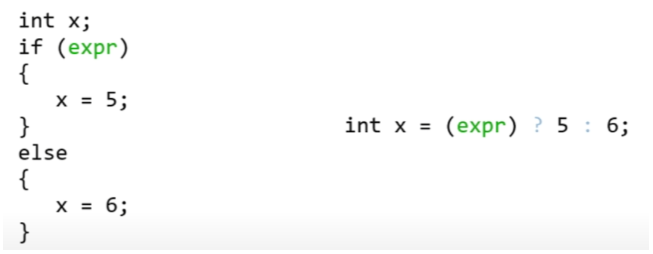
Switch Statement
C's switch statement is a conditional statement that permits enumeration of discrete cases, instead of relying on Boolean expressions



```
int x = GetInt();
switch(x)
{
   case 1:
      printf("One!\n");
      break;
   case 2:
      printf("Two!\n");
      break;
   case 3:
      printf("Three!\n");
      break;
   default:
      printf("Sorry!\n");
}
```

Int this program the user is asked for an int, if they type 1, the program prints "One!" and breaks
If the int = 2 then prints "Two!" and breaks (end the program)
The default response if not 1, 2, or 3 and an integer the program prints Sorry! And quits. If not an integer, repromts

ternary Operator
Unlike all other operators in C—which are either unary or binary operators—the
conditional operator is a ternary operator; that is, it takes three operands
The general format of the conditional operator is
condition ?   expression1 : expression2
where condition is an expression, usually a relational expression, that is evaluated first
whenever the conditional operator is encountered.
If the **result** of the evaluation of condition **is TRUE (that is, nonzero), then expression1 is evaluated** and the result of the evaluation becomes the result of the operation.
If **condition evaluates FALSE (that is, zero), then expression2 is evaluated** and its result becomes the result of the operation.



```
int x;
if (expr)
{
   x = 5;
}
else
{
   x = 6;
}
```

```
int x = (expr) ? 5 : 6;
```

Above: If expression is true assign x the value 5, if expression if false, assign x the value 6

Summary of available Conditionals

If (and if-else || else-if || if-__-else
- (Use Boolean expressions to make decisions)

Switch
- Use discrete cases to make decisions

?
- Use to replace a very simple if-else state
- Ternary operator (int x = (expression) ? 5 : 6;)

Loops

Loops allow your program to execute lines of code repeatedly.

Infinite loop

```
While (true)
{

}
```

The lines of code between {} will execute repeatedly (top to bottom);
Until we break out of the code (as with a break; statement or otherwise kill the program)

While loop

```
While (bool-expr)
{

}
```

If bool = true then code between { } will execute repeatedly (top to bottom);
Until bool = false

Do while Loop

```
do
{

}
while (Boolean-expression)
```

The lines of code between {} will execute once (top to bottom);
Then if the bool = true, program goes back and repeats until bool=false

For Loops

```
For (int i = 0; i < 10; i++)
{

}
```

For loops are used to repeat the body of a loop a specified number of times

- The counter variable (i) is set (0);
- The Boolean-expression is checked (aka condition is checked)
    - The Boolean-expression is checked, and i < 10 = true; the loop executes
    - If the Boolean-expression is checked, an i < 10 = false, the loop does not execute
- The counter variable is incremented and the loop continues will condition is false

Summary of Loops

While

Use when you want a loop to repeat an unknown number of times, or possibly 0 times

Do-while

Use when you want a loop to repeat an unknown number of times, but at least once

For

Use when you want a loop to repeat a discrete number of times, though you may not know
The number at the moment the program is compiled

Magic Numbers

Some of the programs that we've written have some weird numbers thrown in them

e.g. The height of Mario's pyramid is capped at 23

Directly writing constants into our code (referred to as using magic numbers)

Example (Note no function in C called Card, pseudocode used for example)

```
card deal_cards(deck name)
{
        for (int i =0; i < 52; i++)
        {
                // deal the card
        }
}
```

We've got a magic number in the above program? Do you see it?
More importantly do you see a potential problem here?
Particularly if this function is just one of many in a suite of programs that manipulate decks of cards

```
card deal_cards(deck name)
{
        int deck_size = 52
        for (int i =0; i < deck_size; i++)
        {
                // deal the card
        }
}
```

This fixes one problem, but introduces another. It gives some meaning to the constant

Even if globally declared, what if some other function in our suite inadvertently manipulates deck_size?

The deck_size would not constantly = 52

C provides a **preprocessor directive** (also called a **macro**) for creating symbolic constants

```
        #define NAME REPLACEMENT
```

At the time your program is compiled, #define goes through your code and replace NAME with REPLACEMENT
if #include is similar to copy/paste, the #define is analogous to find/repace

```
{
        #define DECKSIZE 52
        for (int i =0; i < DECKSIZE; i++)
        {
                // deal the card
        }
}
```

DECKSIZE will be found and replaced by the compiler with the constant value 52

Functions

Why use functions?
Organization
- Functions help break up a complicated problem into more manageable subparts

Simplification
- Smaller components tend to be easuer to design, implement, and debug

Reusability
- Functions can be recycled, you only need to write them once, but can use them as often as you need

Function Declarations //prototypes

- The first step to creating a function is to declare it. This gives the compiler a heads-up that a user-written function appears in the code
- Function declarations should always go atop your code, before you begin writing main()
- There is a standard from that every function declaration follows

return-type name(argument-list);

return-type is what kind of variable the function will output

name is the name of your function (make it descriptive and concise)

argument-list is the comma-separated set of inputs to your function (type and name)

example
```
int add_two_ints(int a, int b);     //this part is added to the top of your code before int(main)(void)
{
          int sum = a + b;          // calc variable, this is where the function is defined
          Return sum;               // give result back to "add_two_ints", an integer
}
```

Function Calls
Now that you've created a function, time to use it
To call a function, simply pass it appropriate arguments and assign its return value to something of the correct type
To illustrate this, let's have a look adder-1.c

Function calls go into your main function, to specify that something equals the product of the function below and the function calls will specify what values go into function to get the product
E.g
Int z = add_two_ints(x, y); x and y go into the functions to get the product

Function Miscellany
Recall from our discussion of data type that functions can sometimes take no inputs. In that case, we declare the function as having a void argument list.
Recall also that functions sometimes do not have an output. In that case, we declare the function as having a a void return type.

for (init_expression; loop_condition; loop_expression) program statement

The three expressions that are enclosed within the parentheses—init_expression,
loop_condition, and loop_expression—set up the environment for the program loop.
The program statement that immediately follows (which is, of course, terminated by a
semicolon) can be any valid C program statement and constitutes the body of the loop.
This statement is executed as many times as specified by the parameters set up in the for
statement.
The first component of the for statement, labeled init_expression, is used to set
the initial values before the loop begins. In Program 5.2, this portion of the for statement
is used to set the initial value of n to 1.As you can see, an assignment is a valid form of
an expression.
The second component of the for statement the condition or conditions that are
necessary for the loop to continue. In other words, looping continues as long as this condition is satisfied. Once again
referring to Program 5.2, note that the loop_condition of
the for statement is specified by the following relational expression:
n <= 200

This expression can be read as "n less than or equal to 200." The "less than or equal to"
operator (which is the less than character < followed immediately by the equal sign =) is
only one of several relational operators provided in the C programming language. These
operators are used to test specific conditions. The answer to the test is "yes" or, more
commonly, TRUE if the condition is satisfied and "no" or FALSE if the condition is not
satisfied.

The relational operators have lower precedence than all arithmetic operators.

This means, for example, that the following expression

a < b + c

is evaluated as

a < (b + c)

as you would expect.

It would be TRUE if the value of a were less than the value of b + c and FALSE otherwise.

- Pay particular attention to the "is equal to" operator ==
  - and do not confuse its use with the assignment operator =.
  - The expression a == 2 tests if the value of a is equal to 2,
  - whereas the expression a = 2 assigns the value 2 to the variable a.

the program statement that forms the body of the for loop

is repetitively executed as long as the result of the relational test is TRUE, or in this case, as long as the value of n is less than or equal to 200.

This program statement has the effect of adding the value of triangularNumber to the value of n and storing the result back in the value of triangularNumber.

A constant is just an immediate, absolute value found in an expression.

Arrays
Arrays are a fundamental **data structure**

We use arrays to hold values (of the same type) at contiguous memory locations
Arrays are partitioned into small, identically sized blocks of space called **elements**
Each of which can store a certain amount of **data**
Arrays store all of the same data types (such as int or char)
We can access each **element** of an **array** directly via an **index**

In C, the elements of an array are indexed starting at 0

If an array consists of *n* elements, the first elements is located at index 0.
The last elements is located at index (*n* – 1).
(if an array has 50 elements, the array starts at 0 and ends at 49)
C is very lenient. It will not prevent you from going "out of bounds" of an array,
Though is you go out of bounds you will likely get a "segmentation fault" error

Array declarations

Type name[size];

The type is what kind of variable each element of the array will be
The name is what we call the array
The size is how many elements you would like the array to contain

int student_grades [40];

type = int
name = student_grades
size of the array is 40 intergers

if you think of a single element of an array of type data-type the same as you would any other variable of type data-type (which, effectively, it is) then all the familiar operations make sense
bool truthtable[10];

truthtable[2] = false ([2] = the third element of truthtable is false)
if (truthtable[7] == true) (if the 8th element of truth table is true then print "TRUE!"
```
if (truthtable[7] == true)
{
        Printf("TRUE!\n");
}
```
truthtable[10] = true (10 is actually 11 (Start at zero) and is outside of the array
(array goes until [9]

When declaring and initializing an arry simultaneously, there is a special syntax that may be used to fill up the array with its starting values.

These two ways of defining an array produce the same result
// instantiation syntax (when using this, you don't have to specify the size of the array on the left because you put three elements
Bool truthtable[3] = { false, true, true };

// individual element syntax
Bool truthtable[3];

```
truthtable[0] = false;
truthtable[1] = true;
truthtable[2] = true;
```

- You can also use loops to define the values of elements in an array

Exercise to try: Make an array of size [100] where every element = the value of its index

- Arrays can consist of more than a single dimension. You can have as many size specifiers as you wish

  bool battleship[10][10];

- You can think of this as a 10 x 10 grid of cells
- In memory though, it's really just a 100-element one-dimension grid of contiguous cells (aka a list)
- Multi-dimensional arrays are great abstractions to help visualize game boards or other complex representations

- While we can treat individual elements of arrays as variables, we cannot treat entire arrays themselves as variables

- We cannot, for instance, assign one array to another using the assignment operator (not legal C code).
- Instead, we must use a loop to copy over the elements one at a time.

- Say we want to copy the values of elements from one array into another array,
  Then we need to use a loop, copy of each element one at a time

Example:

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

bar = foo;
```

the above example will not work to add 1-5 from array foo to array bar
to do so we need to use a loop, as seen below

```
for(int j = 0; j < 5; j++)
{
   bar[j] = foo[j];
}
```

Recall that most variables in C are **passed by value** is function calls

Passed by value means that we are making a copy of the variable that is being passed in,
The function essentially gets it own local copy of a variable to manipulate

Arrays do not follow this rule. Rather, they are **passed by reference**.
The callee receives the actual array, <u>not</u> a copy of it.

What does the mean when the callee manipulates elements of the array?

Command Line Arguments

So far, all of the programs begin with

```
Int main(void)
{
```

Since we've been collecting user input through in-program prompts,
We haven't needed to modify the declaration of main()

If we want the user to provide data to our program before the program starts running, we need a new form

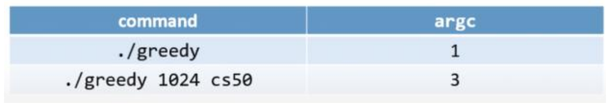
To collect **command-line arguments** from the user, declare main as:

```
int main(int argc, string argv[])
{
```

These two special arguments enable you to know:

- what data the user provided at the command line
- how much data they provided?

argc(argument count) (note: you don't have to use this naming convention - argc
this integer-type variable will store the **number** of command-line arguments the user typed wwhen the program was executed



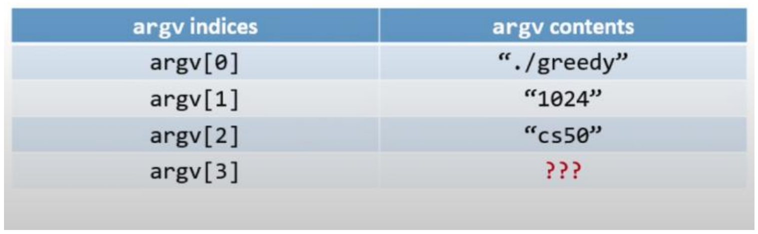
command	argc
`./greedy`	1
`./greedy 1024 cs50`	3

Any amount of white space between the values typed at the cmdline indicates # of argc ints

Argv (argument vector)
Argv (argument vector) vector is another word for an array
This array of strings stores,
one string per element,
the actual text the user typed at the command-line when the program was executed.

The first element of argv is always found at argv[0]
The last element of argv is always found at argv[argc – 1] (start at zero, so subtract 1)

Example
./greedy 1024 cs50 (greedy program is run with 2 argument vectors



argv indices	argv contents
`argv[0]`	“./greedy”
`argv[1]`	“1024”
`argv[2]`	“cs50”
`argv[3]`	???

Note: Everything stored is argv is stored as a string
So argv[1] = “1024” not the integer 1024 string is an array of chars [1][0][2][4]
Should you need the integer 1024 from this array, there is a function to convert strings to integers