# Introduction to C++

# Functions

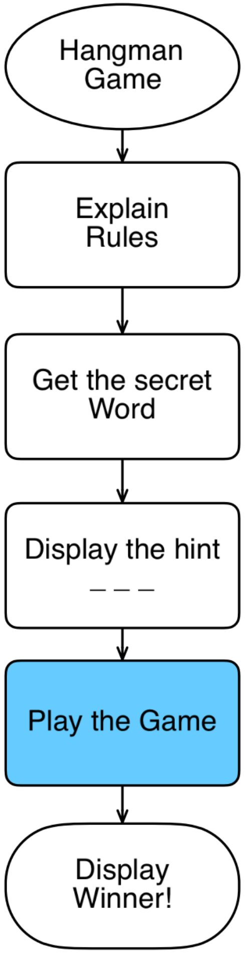
## Topic #2

# Topic #2

- **Solving Problems using Flow Charts and Data Flow Diagrams**

- **Topic #2: Functions**
  - Prototypes vs. Function Definitions
  - Pass by Value, by Reference, by Constant Reference, by Pointer
  - Function Overloading
  - Default Arguments

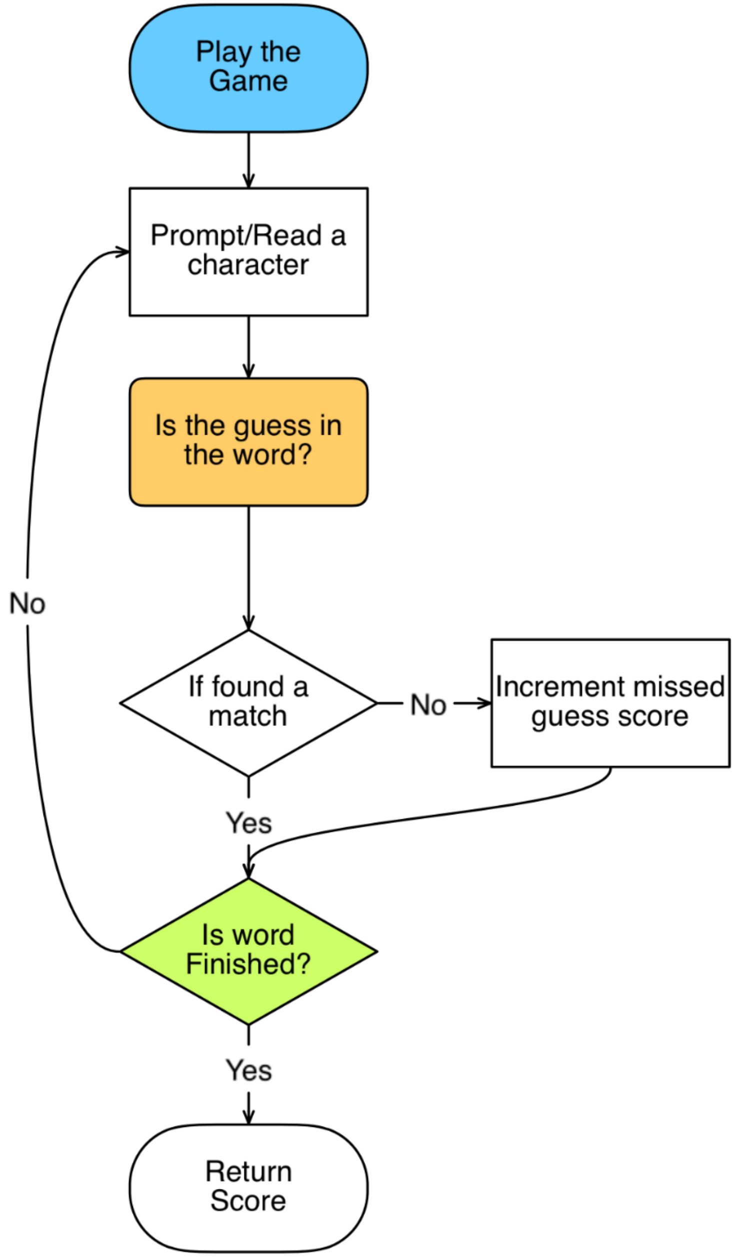
# Designing a Hangman Program:



- Divide the problem into its major tasks
- By having a function perform each major task, we can simplify main and perform the task many times in the same program by simply invoking the function repeatedly.
- The code for the task need not be reproduced every time we need it.
- A function can be saved in a library of useful routines and plugged into any program that needs it.
- For the hangman game, each of these should be a function!

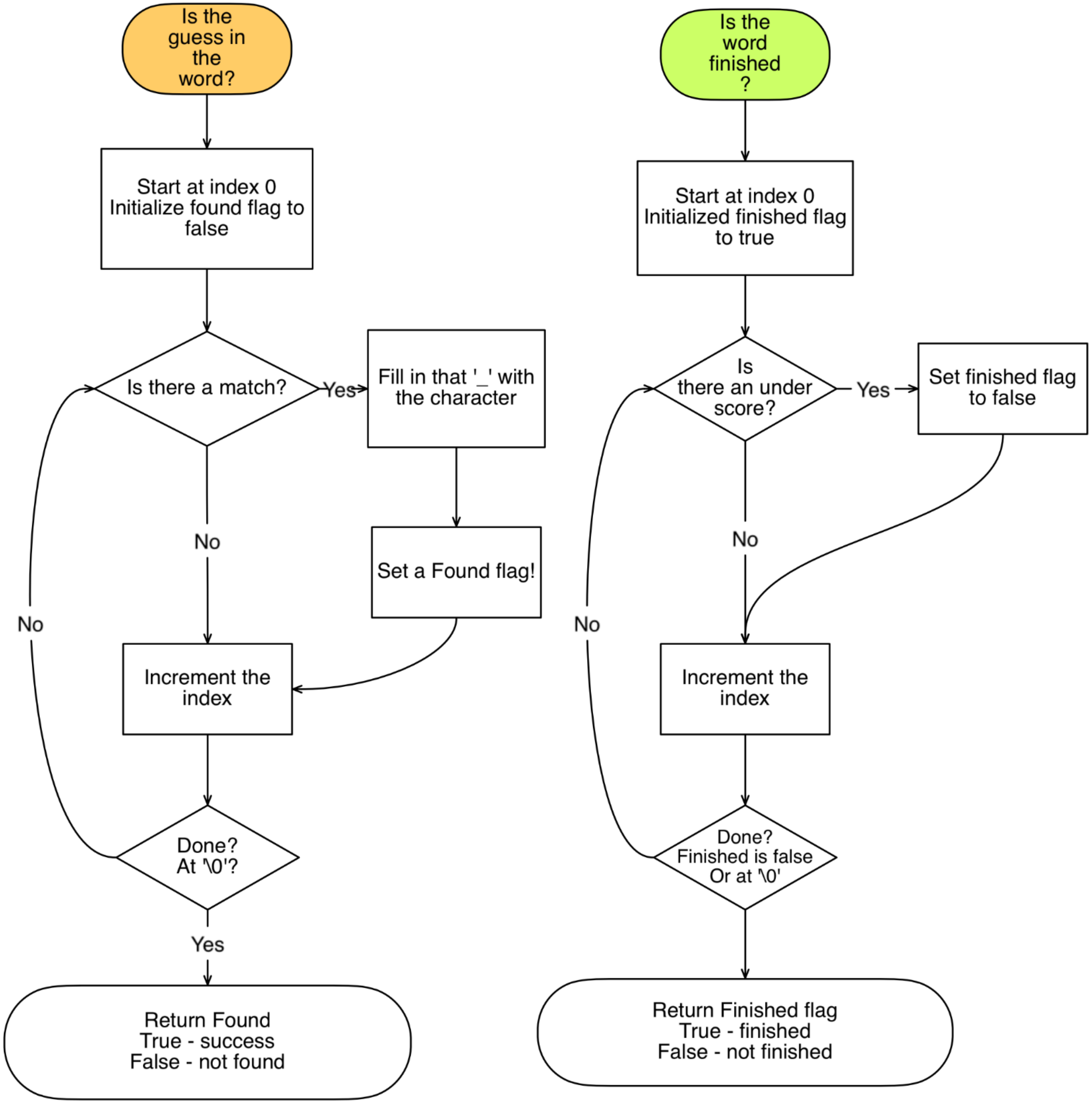
# Functions: What are they?

- We can write our own functions in C++
- These functions can be called from your main program or from other functions
- A C++ function consists of a grouping of statements to perform a certain task
- This means that all of the code necessary to get a task done doesn't have to be in your main program
- You can begin execution of a function by calling the function



# Playing the Game

- Information can be passed from one function to another and data can be returned from a function
- To play the game, we need the secret word to be passed into this function
- We will also be calling the display underscore function to show what is left. That array will also need to be passed in from main
- We will need local variables for the player’s guess and score


Is the guess in the word?
Start at index 0 Initialize found flag to false
Is there a match?
Yes
Fill in that '_' with the character
Set a Found flag!
No
Increment the index
No
Done? At '\0'?
Yes
Return Found True - success False - not found
Is the word finished ?
Start at index 0 Initialized finished flag to true
Is there an under score?
Yes
Set finished flag to false
No
Increment the index
No
Done? Finished is false Or at '\0'
Return Finished flag True - finished False - not finished

# Functions: What are they?

- A function has a name assigned to it and contains a sequence of statements that you want executed every time you invoke the function from your main program!
- Data is passed from one function to another by using arguments (in parens after the function name).
- When no arguments are used, the function names are followed by: "()".

# Functions: Defining Them…

- The syntax of a function is very much like that of a main program.
- We start with a function header:

```
data_type function_name()
{
    <variable definitions>
    <executable statements>
    return variable; //for non-void
}
```

# Functions: Defining Them…

- A function must always be declared before it can be used
- This means that we must put a one-line function declaration at the beginning of our programs which allow all other functions and the main program to access it.
- This is called a **function prototype** (or **function declaration**)
- The function itself can be defined anywhere within the program.

# Functions: Using Them…

- When you want to use a function, it needs to be CALLED or INVOKED from your main program or from another function.
- If you never call a function, it will never be used.
- To call a function we must use the function call operator ()

```
welcome_user();
score = play_the_game(secret_word, underscore_array);
```

- IT IS IMPORTANT to return a value for each function that has a “non void” return type.

# Order of Execution…

- The main program runs first, executing its statements, one after another.
- Even though the functions are declared before the main program (and may also be defined before the main program), they are not executed until they are called.
- They can be called as many times as you wish
- In fact, whenever you find the same operation needs to be done multiple times, write a function and just call it where you need!

# Why write functions?

- Once a function is written and properly tested, we can use the function without any further concern for its validity.
- We can therefore stop thinking about how the function does something and start thinking of what it does.
- It becomes an abstract object in itself - to be used and referred to.
- Look up the concept called unit testing! We can write a function and completely test it out ensuring that it does what is expected before moving to the next function!

# Some details about functions:

- Each function can contain definitions for its own constants and variables (or objects).
- These are considered to be LOCAL to the function and can be referenced only within the function in which they are defined

```
return_data_type some_function()
{
     data_type variable;      //local variable

     return  variable;        //IMPORTANT!!!!!
}
```

# Some details about functions:

```
#include <iostream>
using namespace std;
//Have header comments for each program that you write
int print_asterisk(void);  //Prototype

int main()
{
  int number;                       //local variable
  number = print_asterisk();        //Function call
  ...
}

//Have header comments for each function that you write!!
int print_asterisk ()
{
    int num_asterisk;               //local variable

    cout <<"How many asterisks would you like?\n";
    cin >>num_asterisk;   cin.ignore(100,’\n’);
    return(num_asterisk);  //Returning the number of asterisks!
}
```

# Some details about functions:

- To have a function return a value - you simply say "`return expression`".
  - The expression may or may not be in parens.
- Or, if you just want to return without actually returning a value, just say `return;`
  - Note: return(); is illegal because return is not a function call
  - If you normally reach the end of a function it will automatically return even without a return statement (but you can't return a value without a return statement!)
  - It is dangerous to have many return statements in a function.
  - Structured programming dictates that you should NEVER return from within a loop!
  - Use the conditional expression to determine if a loop should continue and when it should stop.

# Some details about functions:

- For functions that don't return anything, you should preface the declaration with the word "void".
  - Don’t just leave off the return type. If you leave it off, it is expected that the function will return an “int”.
- When using void, it is illegal to have your return statement(s) try to return a value
  - Also notice, that the type of a function must be specified in both the function declaration (prototype) and in the function definition.

# Functions: What are arguments?

- If we want to send information to a function when we call it, we can use arguments

- For example, to play the hangman game, we will have to send information that the main program has to the function
  - Such as the secret word and the resulting underscore word
  - We could have also passed the length of the word to avoid re-calculating that information over and over again!

- We can define functions with no arguments, or with many arguments

# Functions: Prototypes for the Hangman Program

```
//Prototypes
void welcome_message();
void clear_screen();
void get_secret(char secret[]);
void underscores(int length, char array[]);
void display(char array[]);
char read_guess();
bool answer_check(char secret[], char result[]);
void check(char secret[], char result[], char guess);
```

# Functions: Examples using Arguments

```
//Get the scret word
void get_secret(char secret[])
{
        cout << "Please enter your secret word." << endl;
        cin  >> secret;
        cin.ignore(100,’\n’);
}
//Read in the character to be guessed and return it to the user
char read_guess()
{
        cout << "Enter one letter as your guess:";
        cin >> guess;
        cin.ignore(100,’\n’);

        //Capitalize the guess
        guess = toupper(guess);

        return guess;
}
```

# Functions: Examples using Arguments

```
//Turn the resulting array into underscores
void underscores(int length, char result[])
{
        for(int i = 0; i < length; ++i)
        {
                result[i] = '_';
        }
        result[length] = '\0'; //IMPORTANT!
}

//Check to see if we are done!
bool answer_check(char secret[], char result[])
{
        if (strcmp(secret,result) == 0)
                return true;
        return false;
}
```

# Functions: What are arguments?

- Notice that variables are declared in a function heading;
  - these are FORMAL ARGUMENTS
  - they look very much like regular variable declarations, except that they receive an initial value from the function call
- The arguments in the function call (invocation) are called ACTUAL ARGUMENTS.
- When the function call is executed,
  - the actual arguments are conceptually copied into a storage area local to the called function.
  - If you then alter the value of a formal argument, only the local copy of the argument is altered.
  - The actual argument never gets changed in the calling routine.

# Functions: What are arguments?

- C++ checks to make sure that the number and type of actual arguments sent into a function when it is invoked match the number and type of the formal arguments defined for the function.
- The return type for the function is checked to ensure that the value returned by the function is correctly used in an expression or assignment to a variable.

# Functions: Value vs. Reference

- Pass by value brings values into a function (as the initial value of formal arguments)
  - that the function can access but not permanently change the original actual args
  - with arrays, we are passing the starting location of the first element by value
- Pass by reference can bring information into the function or pass information to the rest of the program;
  - the function can access the values and can permanently change the actual arguments!
  - Pass by reference uses the reference operator (&) in the prototype and function header (but not in the function call)
  - Arrays cannot be passed by reference

# Functions: Value vs. Reference

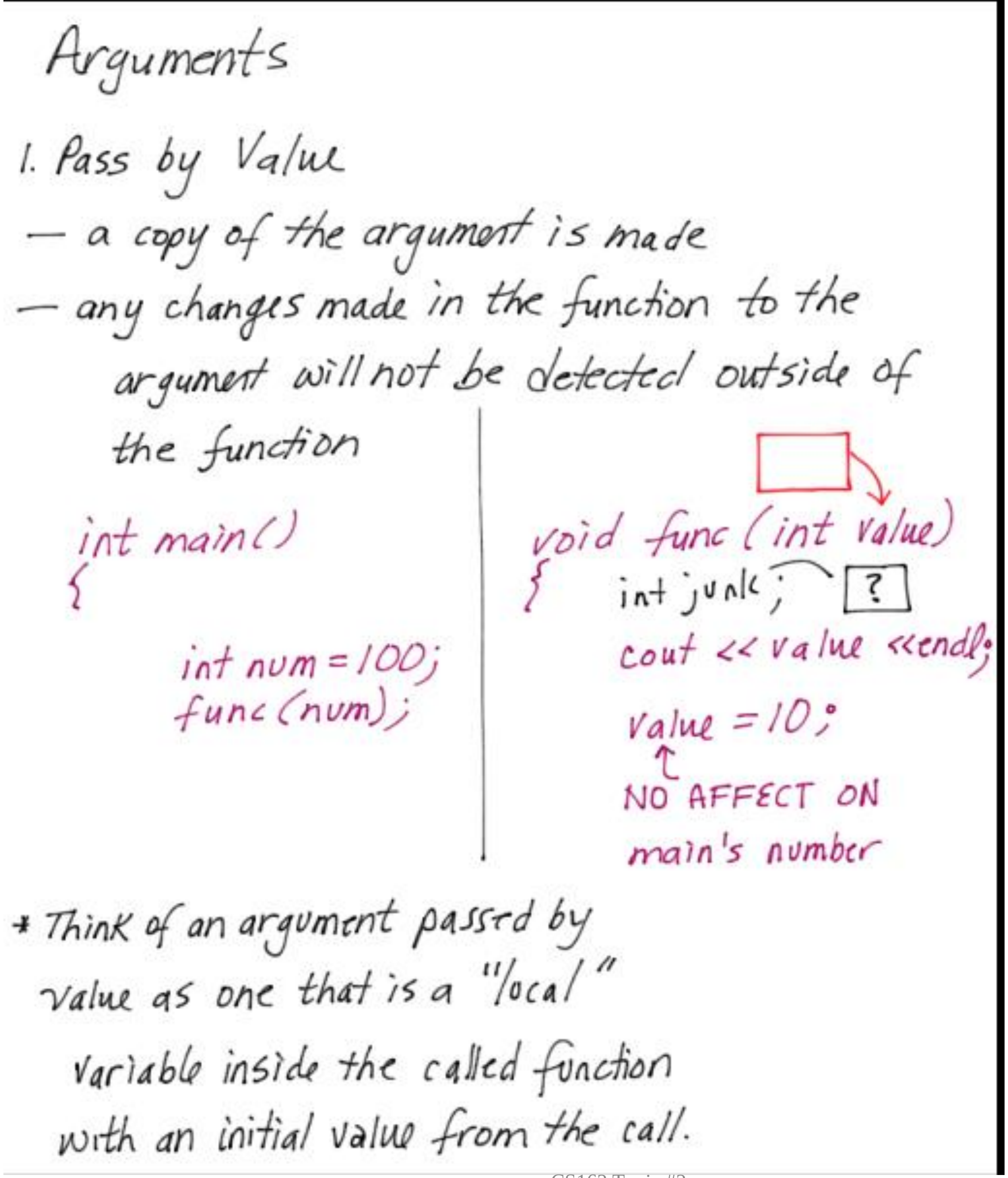
- Pass by value is useful for:

  - passing information to a function

  - allows us to use expressions instead of variables in a function call

  - value arguments are restrained to be modified only within the called function; they do not affect the calling function.

  - can't be used to pass information back, except through a returned value



# Arguments

1. Pass by Value

— a copy of the argument is made

— any changes made in the function to the argument will not be detected outside of the function

```
int main()
{

      int num = 100;
      func(num);
```

```
void func (int value)
{     int junk;  ?
      cout << value <<endl;
      value = 10;
```

NO AFFECT ON main's number

* Think of an argument passed by value as one that is a "local" variable inside the called function with an initial value from the call.

# Functions: Value vs. Reference

- Pass by reference is useful for:

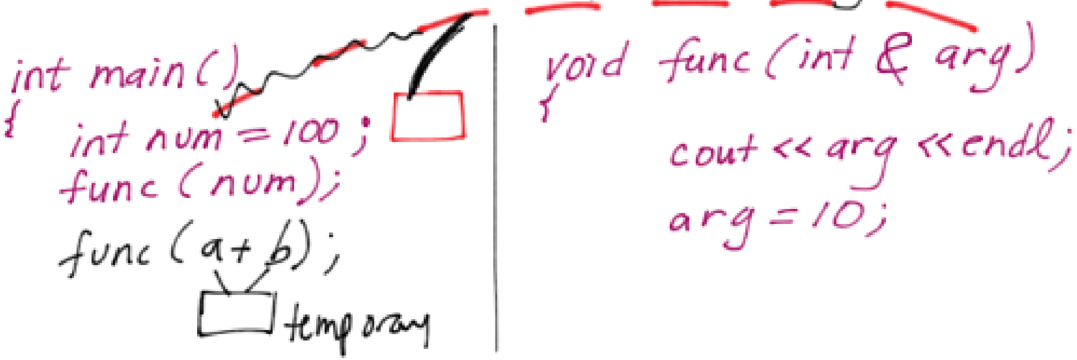
  - allowing functions to modify the value of an argument, permanently

  - requires that you use variables as your actual arguments since their value may be altered by the called function;

  - you can't use constants or literals in the function call!

# Pass by Reference

- creates an alias
- the "address of" the calling routine's value is implicitly sent to the function
- Allows us to "Get Information" from a function without the overhead of returning it



```
int main()
{
  int num = 100;
  func (num);
  func (a+b);
```

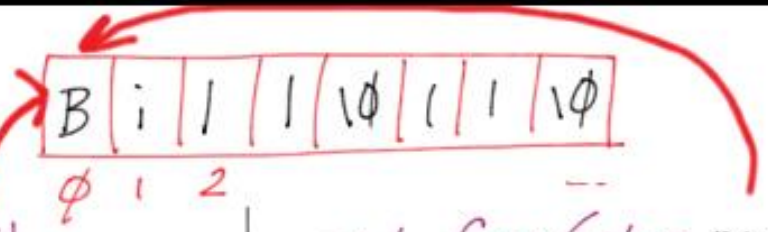
```
void func (int & arg)
{
    cout << arg << endl;
    arg = 10;
```

Any changes made in the called function immediately affect the calling routine's value

(can't be used for passing literals (numbers) or in this case constants)

# Functions: Passing Arrays

- Technically, when an array is an argument, the starting address of the first element is actually passed by value
- This seems like pass by reference because the contents of the array can be altered by the called function
- BUT, it is not possible to pass an array by reference in C++ using the & (reference) operator. If you did, it would mean that the starting address or location of the array could be altered!



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| B | i | l | l | \0 | ( | l | \0 |
| 0 | 1 | 2 | | | | | -- |

```
int main()
{
   char name[21];
   cout << "Enter a name";
   cin.get(name, 21);
   cin.ignore(100, '\n');

   func(name   );
   cout << name   ;
        pass an array:
```

```
void func(char name[])
{
      cout << array <<endl;

      //what about:
      cout << "Re-enter: ";
      cin.get(array, 21);
      cin.ignore(100, '\n');
      cout << array;
```

# What kind of args to use?

- Use a call by reference if:

  1) The function is supposed to provide information to some other part of the program. Like returning a result and returning it to the main.

  2) They are OUT or both IN and OUT arguments.

  3) In reality, use them WHENEVER you don’t want a duplicate copy of the arg...

# What kind of args to use?

- Use a call by value:

  1) The argument is only to give information to the function - not get it back

  2) They are considered to only be IN parameters. And can't get information back OUT!

  3) You want to use an expression or a constant in function call.

  4) In reality, use them only if you need a complete and duplicate copy of the data

# Swapping to arrays – example

```
#include <cstring>
void sort_two(char first[], char second[]) {
 cout <<“Please enter two words: “;
 cin.get(first,20, ‘ ‘);    cin.get();
 cin.get(second,20, ‘\n’);
 cin.get();   //eat the carriage return;
 if (strcmp(first, second) > 0) {
     char temp[20];
     strcpy(temp,first);
     strcpy(first, second);
     strcpy(second,temp);
 }
}
```

# We’d call the function by saying:

```
#include <string.h>
void sort_two(char first[], char second[]);
void main() {
 char str1[20], str2[20];

 sort_two(str1, str2);
 cout <<str1 <<‘ ‘ <<str2 <<endl;

 //what would happen if we then said:
 sort_two(str2, str1);
 cout <<str1 <<‘ ‘ <<str2 <<endl;
}
```

# Introduction to C++

## Structures

# What is a Structure

- A structure is a way for us to group different types of data together under a common name
- With an array, we are limited to having only a single type of data for each element...
  - think of how limiting this would be if we wanted to maintain an inventory
  - What would happen if we wanted to represent an inventory of information with a product name, barcode, description, price and distributor
  - we'd need a separate array for each product's name, another for each product's price, and yet another for each barcode!
    - char name[41];
    - char description[131];
    - float price;
    - char distributor[113];
  - But, how could you have more than one product?

# What is a Structure

- With a structure, on the other hand, we can group each of these under a common heading
  - So, if each product can have a description, a price, a cost, and a barcode....a single structure entity can consist of an array of characters for the description, two floats for the price and cost, and an int for the barcode
  - Now, to represent the entire inventory we can have an array of these “products”

# Why would we use a Structure

- Some people argue that with C++ we no longer need to use the concept of structures
- And, yes, you can do everything that we will be doing with structures, with a “class” (which we learn about next week!)
- My suggestion is to use structures whenever you want to group different types of data together, to help organize your data

# How do you define a Structure?

```
struct product ← This is the “tag name”
{
        char item[20];  ← Each of these is
        float cost;        a member
        float price;
        int barcode;
 };  ← The semicolon IS required!
```

- In this example, `item`, `price`, `cost` and `barcode` are member names. `product` is the name of a new derived data type consisting of a character array, two real numbers, and an integer.

# Once we define a structure…

- Each component of a structure is called a member and is referenced by a member name (identifier).
    - Structures differ from arrays in that members of a structure do not have to be of the same type.
    - And, structure members are not referenced using an index.
- Once a structure is defined,
    - We have created a new “data type”
    - This is a “specification” that can then be used to create variables of type product
    - Each variable created will have memory for each of the members outlined in the previous “specification”
    - When a structure is specified…NO MEMORY IS ALLOCATED YET
    - We are simply specifying what memory we want ONCE a variable of this type has been created
    - Now we can create zero or more variables of this time
    - We can even create arrays of this type!

# Where do we define Structures?

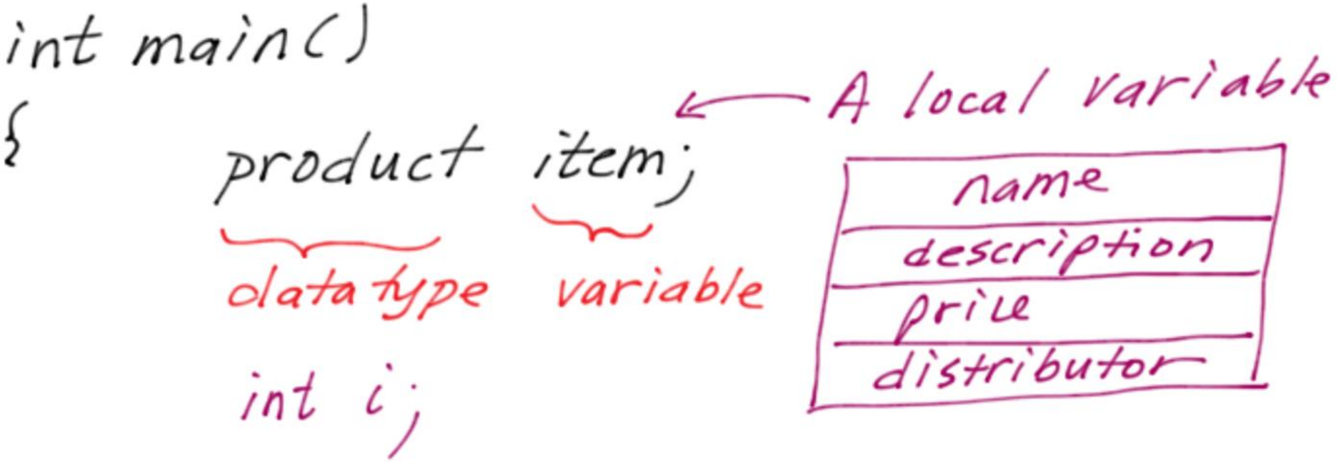
- We typically define structures “globally”
  - This means they are placed outside of the main
- We do this because structures are like a “specification” or a new “data type”
  - Which means that we would want all of our functions to have access to this way to group data, and not just limit it to some function by defining it to be local.
  - This is OK becaue a structure declaration is a “specification” and no memory is allocated yet.
  - So there are **NO side effects** by placing the structure definition globally.
  - This is NOT the case if you were to create a variable of that structure type globally.

# How do you define variables of a Structure?

- Once your have declared this new derived data type, you can create variables (or “object”) which are of this type (just like we are used to).
- For example, we could say from main:

```
    product one_item;
```

- If this is done in a function, then one_item is a local variable...



# How do you define variables of a Structure?

- By saying:

```
product one_item;
```

- From this statement, one_item is the variable (or object)
  - We know that we can define a product which will have the components (members) of the item name, the cost, the price, and the bar code.

  - Just think of product as being a type of data which consists of an array of characters, two real numbers, and an integer.
  - And, now we can easily pass this product to functions very easily
    - Although you will always want to pass structures by value and NEVER return them from functions – *but more about that later!*

# How do you define variables of a Structure?

- By saying:

```
product one_item;      //one_item is the variable!
```

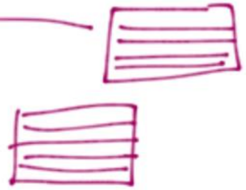
  - To access a structure variable's components, we use dots (the direct member access operator) between the struct **VARIABLE** and the **member's name**:

```
 cin >> one_item.item;               //an array of chars
cout << one_item.item[0];            //1st character...

 cin >> one_item.price               //a float
 cin >> one_item.barcode             //an int
```

# How do you define variables of a Structure?

## Accessing Members

```
product another;
product item;
cin.get(item.name, 41, '\n');
```



```
object.member
```

variable or object of type struct

Direct member access operator

```
Variable.Member
```

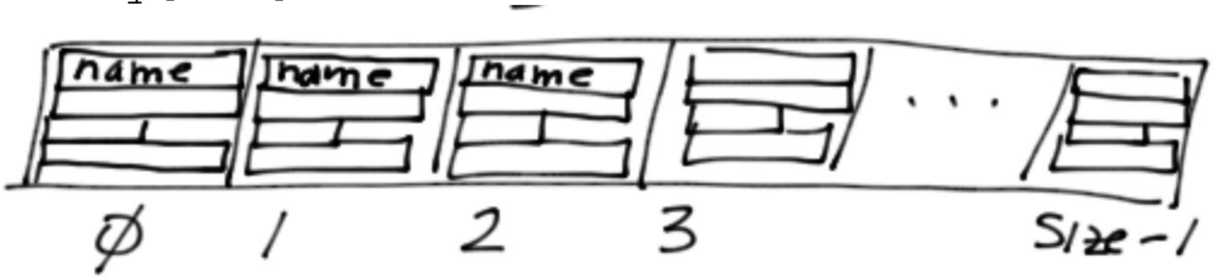
# What operations can be performed?

- Just like with arrays, there are very few operations that can be performed on a complete structure
- We **can't** read in an entire structure at one time, or write an entire structure, or use any of the arithmetic operations...
  - We must work individually with each member as show on the previous slide
- We can use assignment, to do a "memberwise copy" copying each member from one struct variable to another
  - One_item = another_item;
  - However, this won't work correctly once we progress into Topic #2
  - Therefore, each and every member must be copied to assign one structure to another correctly
- But we CAN pass a structure to a function as a complete entity
  - Minimizing the number of arguments needed
  - BUT MAKE SURE to use pass by reference!!!

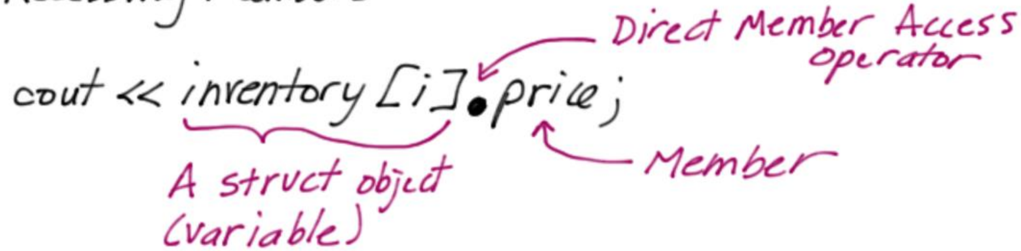
# How do you define arrays of Structures?

- But, for structures to be meaningful when representing an inventory
  - we may want to use an array of structures
  - where every element represents a different product in the inventory
- For a store of 100 items, we can then define an array of 100 structures:

```
product inventory[100];
```



Accessing Members:



```
cout << inventory[i].price;
```

# Working with arrays of Structures

- Notice, when we work with arrays of any time OTHER than an array of characters,
  - we don’t need to reserve one extra location
  - because the terminating nul doesn’t apply to arrays of structures, (or an array of ints, or floats, ...)
  - so, we need to keep track of how many items are actually stored in this array (10, 50, 100?)
  - This means we will need an integer counter when working with arrays of structures to keep track of how full the array actually is!

```
product inventory[100];
int num_items = Ø;
```

# Working with arrays of Structures

- So, once an array of structures is defined, we can access each element via indices:

```
product inventory[100];
int num_items=0, i = 0;

//get the first product’s info
cin.get(inventory[i].item, 21);
cin.ignore(100,’\n’);
cin >>inventory[i].price
     >>inventory[i].cost
     >>inventory[i].barcode;
cin.ignore(100,’\n’);

++num_items;
```

# How do you pass Structures to functions?

- To pass a structure to a function, we must decide whether we want call by reference or call by value
- By reference, we can pass 1 store item:

```
return_type function(product & arg);
```

- Or, we can pass an array of items
  - When an array is passed, the location of the first element is passed by value

```
return_type function(product arg[]);
```

- NEVER pass a structure by value
- NEVER return a structure by value

# Passing Structures to functions

1. Prototype:

```
void inputinventory (product & an_item);
```

Never pass a struct by value.

2. Function Call:

```
product item;
input_inventory (item);
```

an object or variable of type struct product

3. Function Implementation:

```
void input_inventory(product & object)
{       cout << "Enter a name: ";
        cin.get(object.name, 41);
        cin.ignore(100, '\n');
//etc
```

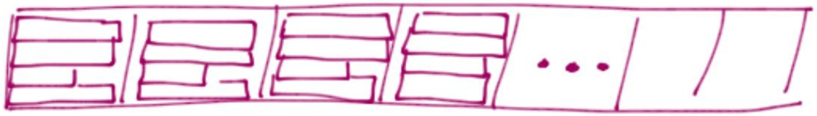
# Passing Arrays of Structures

```
product inventory[100];
int num_items = Ø;
```

```
for (int i=Ø; i<100; ++i)
    input_inventory (inventory [i]);
```

Passing one object by Ref.

one structure instance



To Pass an entire array:

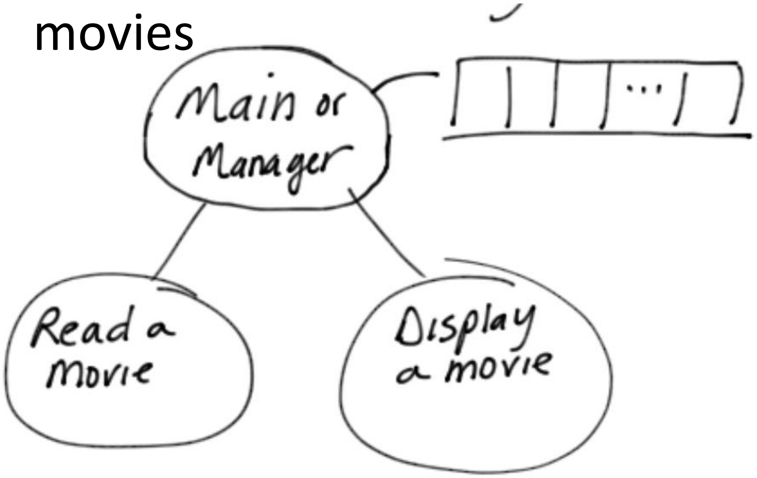
```
prototype:  void display_all (product array [], int num);
call:  display_all (inventory, i);
```

array

the number of elements in the array.

# Developing Code using Structures

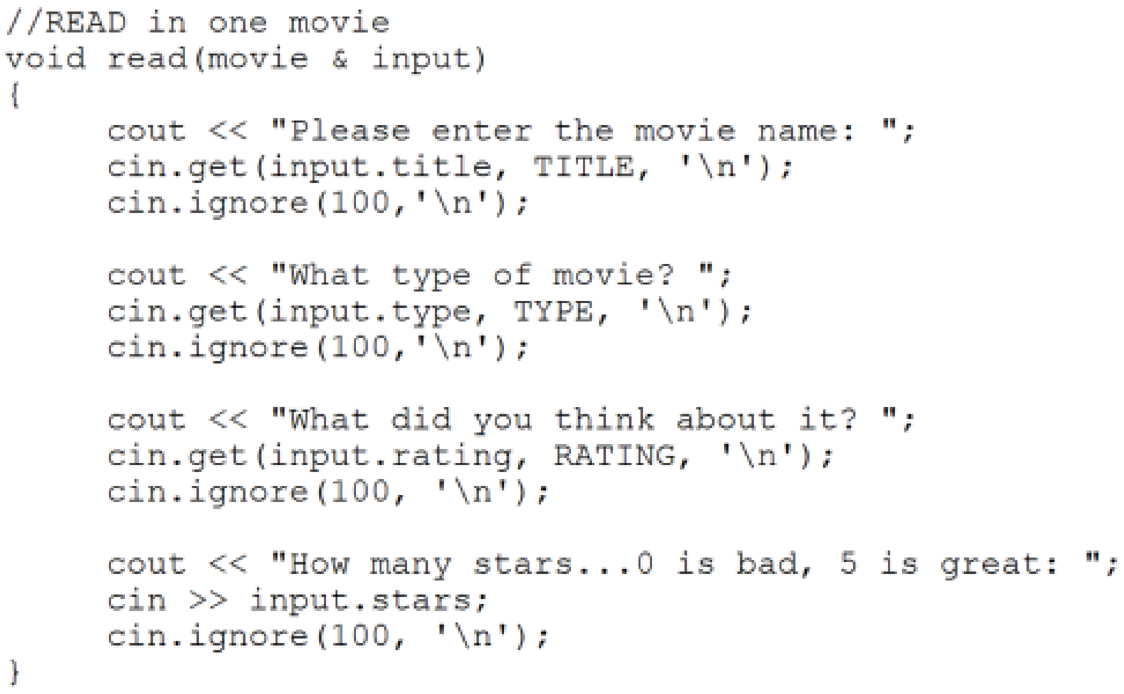
- Goal: Manage a list of movies
- Step #1: Create a structure to manage just one movie
- Step #2: Test this for a single movie by creating a variable of this type in main
- Step #3: Then create an array of movies



```
//Constants for the sizes of arrays
const int TITLE = 21;
const int TYPE = 16;
const int RATING = 131;

struct movie
{
    int stars;   //5 stars is great
    char title[TITLE];
    char type[TYPE];
    char rating[RATING];
};
```

# Developing Functions using Structures

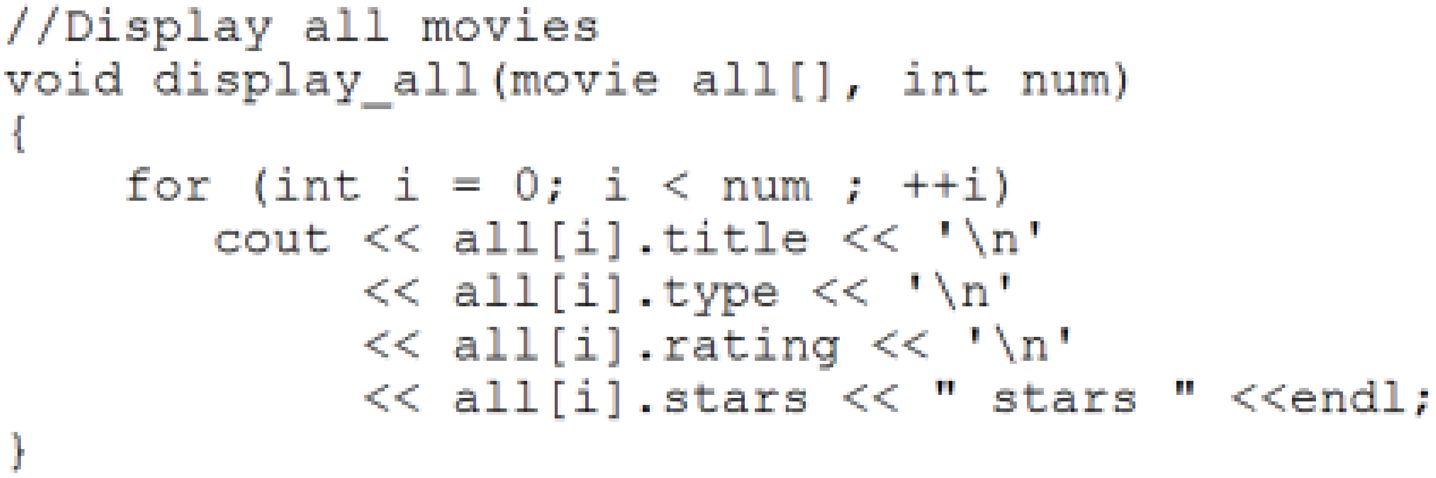
- Now develop a function to read a movie a movie
  - Test out that the code works with just one movie before progressing
- Prototype: **void read(movie & to_read_in);**



```
//READ in one movie
void read(movie & input)
{
     cout << "Please enter the movie name: ";
     cin.get(input.title, TITLE, '\n');
     cin.ignore(100,'\n');

     cout << "What type of movie? ";
     cin.get(input.type, TYPE, '\n');
     cin.ignore(100,'\n');

     cout << "What did you think about it? ";
     cin.get(input.rating, RATING, '\n');
     cin.ignore(100, '\n');

     cout << "How many stars...0 is bad, 5 is great: ";
     cin >> input.stars;
     cin.ignore(100, '\n');
}
```

# Developing Functions using Structures

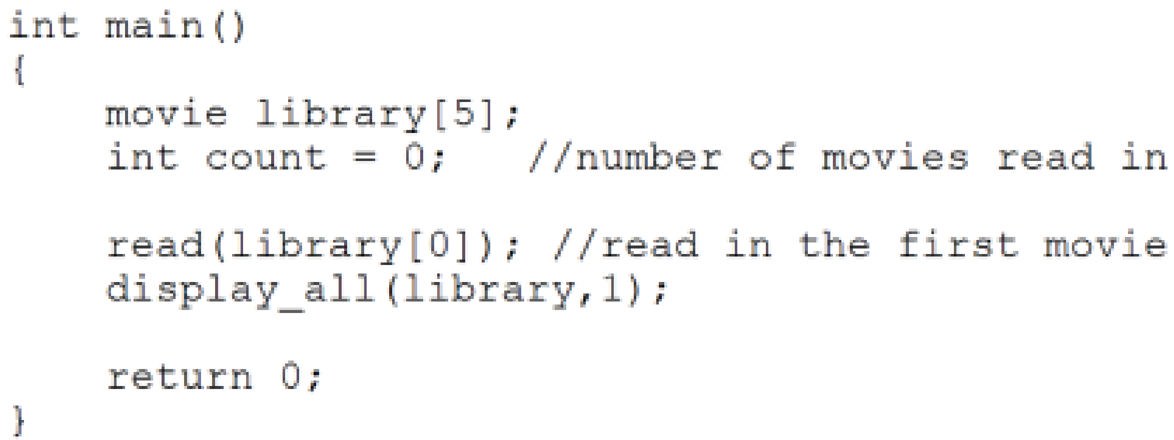
- Now develop a function to display all movies
  - To display all, we will need to send in the integer count of the number of movies read in, otherwise we will display garbage!
- Prototype: **void display_all(movie array[], int number_movies);**



```
//Display all movies
void display_all(movie all[], int num)
{
    for (int i = 0; i < num ; ++i)
        cout << all[i].title << '\n'
             << all[i].type << '\n'
             << all[i].rating << '\n'
             << all[i].stars << " stars " <<endl;
}
```

# Using Structures from Main

- Let’s Let’s experience calling these functions from main:



```
int main()
{
    movie library[5];
    int count = 0;   //number of movies read in

    read(library[0]); //read in the first movie
    display_all(library,1);

    return 0;
}
```

- For practice, modify this main to read in as many as the user wants (up to 5 movies) until they are done, and then display all movies!