Introduction to C++

Functions

Today's Agenda

- **Topic #2: Functions**
  - Prototypes vs. Function Definitions
  - Pass by Value, by Reference, by Constant Reference, by Pointer
  - Function Overloading
  - Default Arguments
- **Structures and Dynamic Memory**
  - Structures
  - Pointers
  - Dynamic Memory Allocation/Deallocation

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

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>
}
```

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 ()

```
some_variable = pow (x, 3);
```

Functions: Calling pow...

- When we call a function, we are temporarily suspending execution of our main program (or calling routine) and executing the function.
- `pow` takes two values as arguments (`x` and `3`), called **actual arguments** and returns to the calling routine the result (a floating point value)

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

Why write functions?

- By having a function perform the task, we can 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. (like we have seen with the pow function)

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.

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

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

    }
```

Some details about functions:

```
#include <iostream.h>
int print_asterisk(void);
int main(){
  int number;              //local variable
  number = print_asterisk();
  ...
  }

int print_asterisk () {
    int num_asterisk;            //local variable

    cout <<"How many asterisks would you like?\n";
    cin >>num_asterisk;
    return(num_asterisk);
  }
```

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).
- If you normally reach the end of a function (the function's closing "}"), its just like saying `return;` and no value is returned.

Some details about functions:

- For functions that don't return anything, you should preface the declaration with the word "void".
- 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 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, when we supplied two items within the parentheses for the pow function -- these were arguments that were being passed to the function pow!
- We can define functions with no arguments, or with many arguments

Functions: What are arguments?

- **If we go back to our example of converting inches to millimeters...**
  - if we write a function to perform the calculations, we would need to somehow send to the function the number of inches to convert
  - this can be done by passing in the number of inches as an argument
  - and receiving the number of millimeters back as the returned value

Functions: What are arguments?

- For example, from our main program we could say:

```
float convert (float inches);  //prototype

void main() {
    float in;     //local variable to hold # inches
    float mm;     //local variable for the result
    cout <<"Enter the number of inches: ";
    cin >>in;
    mm = convert (in);  //function call
    cout <<in <<" inches converts to " <<mm <<"mm";
}
```

Functions: What are arguments?

- Then, to implement the function we might say:

```
float convert (float inches) {
    float mils;          //local variable
    mils = 25.4 * inches;
    return mils;         //return (mils);
}
```

Functions: What are arguments?

- Notice that we can have arguments to functions!
- These must be in the function header for both the function declaration (prototype) and function definition.
- In this example, `inches` is a variable...which is a argument because it is defined in the function header.

Functions: What are arguments?

- When you call convert,
  - you are establishing an association between the main program's `in` variable
  - and the function's `inches` variable;
  - this function does some calculations,
  - and returns a real number which is stored in the calling routines `mm` variable.

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.

Functions: What are 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: What are arguments?

- When we deal with FORMAL VALUE ARGUMENTS...
  - the calling actual argument values cannot be modified by the function.
  - This allows us to use these functions, giving literals and constants as arguments without having conflicts.
  - This is the default way of doing things in C++.

Let's write a function to sum two numbers:

```
int sumup(int first, int second); //function prototype
void main() {
    int total, number, count;
    total = 0;
    for (count = 1; count <= 5; count++) {
        cout <<" Enter a number to add: ";
        cin >>number;
        total = sumup(total, number);     //function call
    }
    cout <<" The result is: " <<total <<endl;
}
int sumup(int first, int second) {       //definition
    return  first + second;
}
```

Functions: Value vs. Reference

- Call 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
- Call 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!

Functions: Value vs. Reference

- Call 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

Functions: Value vs. Reference

- Call 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!

Example of call by reference:

```
void convert (float inches, float & mils);
int main() {
        float in;     //local variable to hold # inches
        float mm;     //local variable for the result
        cout <<"Enter the number of inches: ";
        cin >>in;
        convert (in, mm);   //function call
        cout <<in <<" inches converts to " <<mm <<"mm";
        return 0;
   }
void convert (float inches, float & mils) {
        mils = 25.4 * inches;
   }
```

Example of call by reference:

```
void swap (int & a, int & b);
int main() {
   int i=7, j = -3;
   cout <<"i and j start off being equal to :" <<i
        <<" & " <<j <<'\n';
   swap(i,j);
   cout <<"i and j end up being equal to    :" <<i
        <<" & " <<j <<'\n';
   return 0;
}
void swap(int &c,int&d) {
   int temp = d;
   d = c;
   c = temp;
}
```

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

Writing a function to work with strings:

```
#include <cstring>
void sort_two() {
   char first[20], second[20];
   cout <<"Please enter two words: ";
   cin.get(first,20, ' ');
   cin.get();       //don't forget this part!
   cin.get(second,20, '\n');
   cin.get();       //eat the carriage return;
   if (strcmp(first, second) < 0)
       cout <<first <<' ' <<second <<endl;
   else
       cout <<second <<' ' <<first <<endl;
}
```

Change the function to have args:

```
#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);
   }
}
```

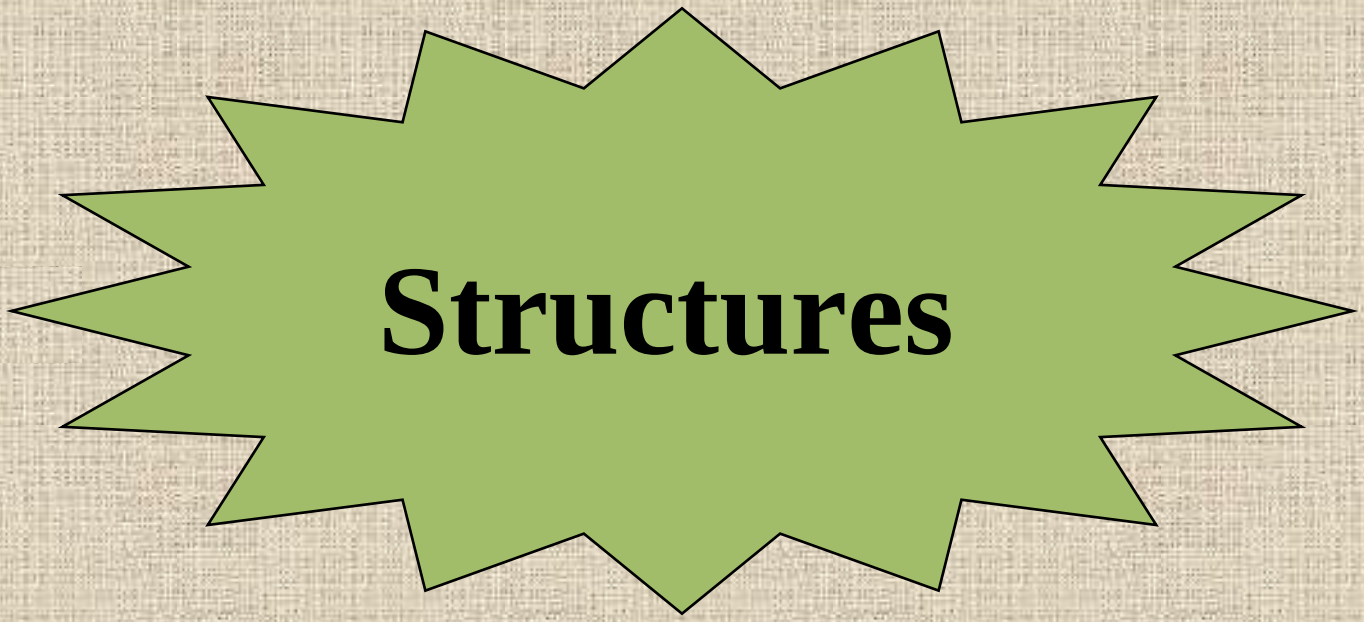
We'd call the function by saying:

```
#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++



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
  - we'd need a separate array for each product's name, another for each product's price, and yet another for each barcode!

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?

- 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

How do you 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.

How do you define a Structure?

- A structure might look like:

```
struct storeitem {
        char item[20];
        float cost;
        float price;
        int barcode;
   };
```

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

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):

```
storeitem 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:

```
storeitem 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 of the item name, the cost, the price, and the bar code.
  - Just think of storeitem as being a type of data which consists of an array of characters, two real numbers, and an integer.

How do you define variables of a Structure?

- By saying:

```
storeitem one_item;
```

  - To access a structure variable's components, we use dots between each field identifiers:

```
one_item.item //an array of chars
one_item.item[0] //1st character...

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

How do you define variables of a Structure?

- We can work with these variables in just the same way that we work with variables of a fundamental type:
- To read in a price, we can say:

  cin >>one_item.price;
- To display the description, we say:

  cout <<one_item.item;

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 can use assignment, to do a "memberwise copy" copying each member from one struct variable to another

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:

```
storeitem inventory[100];
```

How do you define 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?)

How do you define arrays of Structures?

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

```
storeitem inventory[100];
int inv_count=0;
//get the first product's info
cin.get(inventory[inv_count].item, 21);
cin >>inventory[inv_count].price
    >>inventory[inv_count].cost
    >>inventory[inv_count].barcode;
++inv_count;
```

How do you pass Structures to functions?

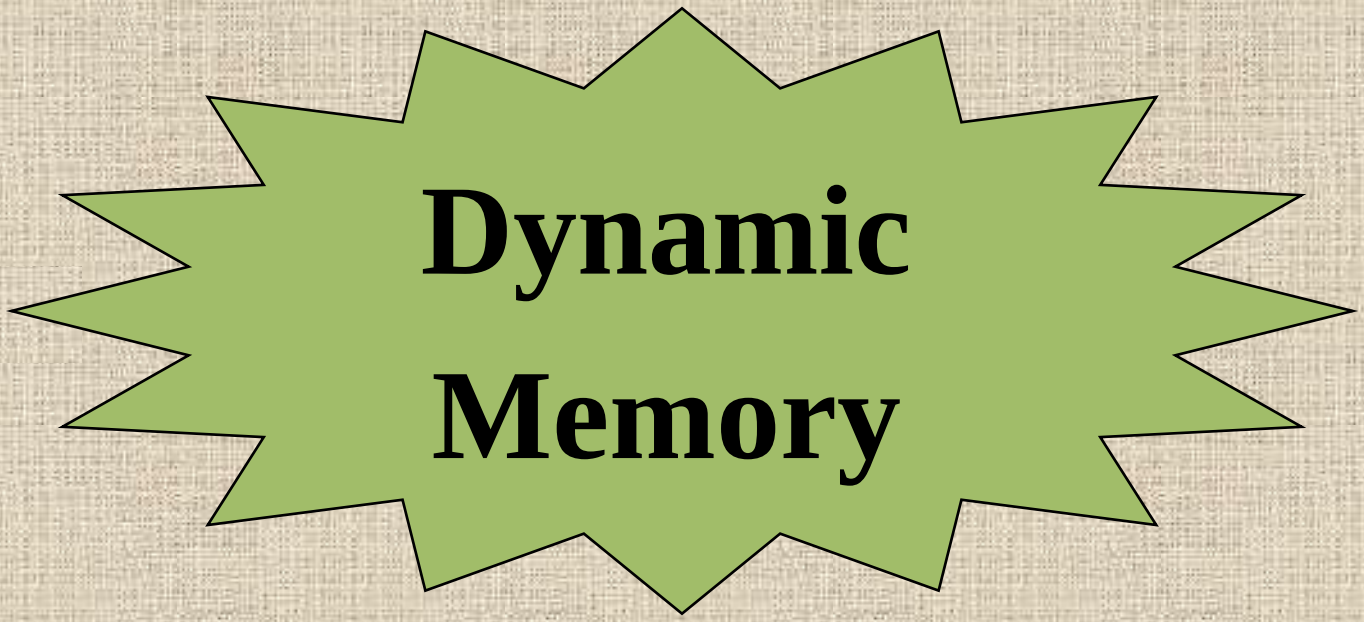
- To pass a structure to a function, we must decide whether we want call by reference or call by value
- By reference, we can pass 1 store item:

```
return_type function(storeitem & arg);
//or an array of store items:
return_type function(storeitem arg[]);
```

- By value, we can pass 1 store item:

```
storeitem function(storeitem arg);
```

Introduction to C++



Pointers

- In C++, a pointer is just a different kind of variable.
- This type of variable points to another variable or object
  - (i.e., it is used to store the memory address of another variable nor an object).
  - Such pointers must first be defined and then initialized.
  - Then, they can be manipulated.

Pointers

- A pointer variable is simply a new type of variable.
  - Instead of holding an int, float, char, or some object's data....it holds an address.
  - A pointer variable is assigned memory.
  - the contents of the memory location is some address of another “variable”.
  - Therefore, the value of a pointer is a memory location.

Pointers

- We can have pointers to (one or more)
  - integers
  - floating point types
  - characters
  - structures
  - objects of a class
- Each represents a different type of pointer

Pointers

- We define a pointer to an integer by:

```
int * ptr; //same as int *ptr;
```

- Read this variable definition from *right to left:*
  - ptr is a pointer (that is what the * means) to an integer.
  - this means ptr can contain the address of some other integer

Pointers

- At this point, you may be wondering why pointers are necessary.
- They are essential for allowing us to use data structures that grow and shrink as the program is running.
  - *linked lists, trees, or graphs*
    - We are no longer stuck with a fixed size array throughout the lifetime of our program.

Pointers

- But first,
  - we will learn that pointers can be used to allow us to set the size of an array at run-time versus fixing it at compilation time;
  - if an object is a list of names...then the size of that list can be determined dynamically while the program is running.
  - This cannot be accomplished in a user friendly way with simple arrays!

Defining Pointers

- So, what are the data types for the following variables?

```
int *ptr1, obj1;    //watch out!
char *ptr2, *ptr3;
float obj2, *ptr4;
```

- What are their initial values (if local variables)? -- *yes, garbage* --

Defining Pointers

- The best initial value for a pointer is
  - zero (address zero),
  - also known as NULL (this is a #define constant in the iostream library for the value zero!)
  - The following accomplish the same thing:

```
int *ptr1 = NULL;
int *ptr2 = 0;
int *ptr3 (0);
```

Defining Pointers

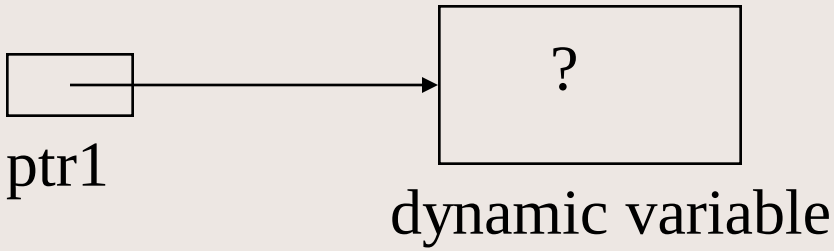
- You can also initialize or assign the address of some other variable to a pointer,
  - using the address-of operator

```
int variable;
int *ptr1 = &variable;
```

Allocating Memory

- Now the interesting stuff!
- You can allocate memory dynamically (as our programs are running)
  - and assign the address of this memory to a pointer variable.

```
int *ptr1 = new int;
```



```
int *ptr1 = new int;
```

- The diagram used is called a
  - pointer diagram
  - it helps to visualize what memory we have allocated and what our pointers are referencing
  - notice that the dynamic memory allocated is of size int in this case
  - and, its contents is uninitialized
  - new is an operator and supplies back an address of the memory set allocated

Dereferencing

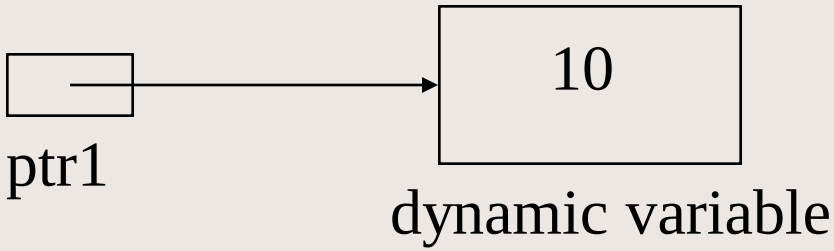
- Ok, so we have learned how to set up a pointer variable to point to another variable or to point to memory dynamically allocated.
- But, how do we access that memory to set or use its value?
- By **dereferencing** our pointer variable:

```
*ptr1 = 10;
```

Dereferencing

- Now a complete sequence:

```
int *ptr1;
ptr1 = new int;
*ptr1 = 10;
...
cout <<*ptr1;  //displays 10
```



Deallocating

- Once done with dynamic memory,
  - we must deallocate it
  - C++ does not require systems to do "garbage collection" at the end of a program's execution!
- We can do this using the delete operator:

```
delete ptr1;
```

this does not delete the pointer variable!

Deallocating

- Again:

  this does not delete the pointer variable!

- Instead, it deallocates the memory referenced by this pointer variable
  - It is a no-op if the pointer variable is NULL
  - It does not reset the pointer variable
  - It does not change the contents of memory
  - *Let's talk about the ramifications of this...*

Allocating Arrays

- But, you may be wondering:
  - Why allocate an integer at run time (dynamically) rather than at compile time (statically)?
- The answer is that we have now learned the mechanics of how to allocate memory for a single integer.
- Now, let's apply this to arrays!

Allocating Arrays

- By allocating arrays dynamically,
  - we can wait until run time to determine what size the array should be
  - the array is still “fixed size”...but at least we can wait until run time to fix that size
  - this means the size of a dynamically allocated array can be a variable!!

Allocating Arrays

- First, let's remember what an array is:
  - the name of an array is **a constant address to the first element in the array**
  - So, saying char name[21];

    means that name is a constant pointer who's value is the address of the first character in a sequence of 21 characters

Allocating Arrays

- To dynamically allocate an array
  - we must define a pointer variable to contain an address of the element type
- For an array of characters we need a pointer to a char:

```
char *char_ptr;
```

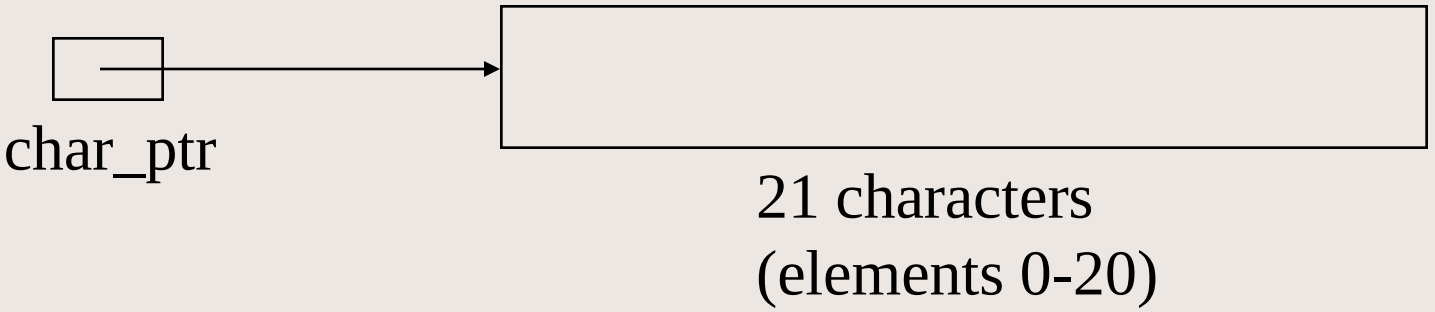
- For an array of integers we need a pointer to an int:

```
int *int_ptr;
```

Allocating Arrays

- Next, we can allocate memory and examine the pointer diagram:

```
int size = 21; //for example
char *char_ptr;
char_ptr = new char [size];
```



Allocating Arrays

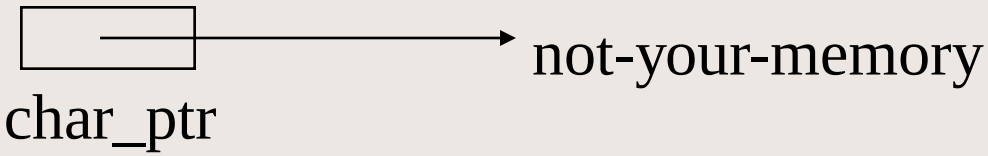
- Some interest thoughts:
  - the pointer diagram is identical to the pointer diagram for the statically allocated array discussed earlier!
  - therefore, we can access the elements in the exact same way we do for any array:

```
char_ptr[index] = 'a';     //or
cin.get(char_ptr,21,'\n');
```

Allocating Arrays

- The only difference is when we are finally done with the array,
  - we must deallocate the memory:

```
delete [] char_ptr;
```



It is best, after doing this to say: char_ptr = NULL;

Allocating Arrays

- One of the common errors we get
  - once allocating memory dynamically
  - is a segmentation fault
  - it means you have accessed memory that is not yours,
    - you have dereferenced the null pointer,
    - you have stepped outside the array bounds,
    - or you are accessing memory that has already been deallocated

Pointer Arithmetic

- When we use the subscript operator,
  - pointer arithmetic is really happening
  - this means the following are equivalent:

```
ptr1[3]  ==  *(ptr1+3)
```

  - This means the subscript operator adds the value of the index to the starting address and then deferences the quantity!!!

Dynamic Structures

- Let's take these notions and apply them to dynamically allocated structures
- What if we had a storeitem structure, how could the client allocate an item dynamically?

```
storeitem *ptr = new storeitem;
```

- Then, how would we access the item?

```
*ptr.item
```

? Nope! WRONG

Dynamic Structures

- To access a member of a struct, we need to realize that there is a “precedence” problem.
- Both the dereference (*) and the member access operator (.) have the same operator precedence....and they associate from right to left
- So, parens are required:

`(*ptr).item` Correct (but ugly)

Dynamic Structures

- A short cut (luckily) cleans this up:

  `(*ptr).item` Correct (but ugly)

  Can be replaced by using the indirect member access operator (->) ... it is the dash followed by the greater than sign:

  `ptr->item` Great!

Dynamic Structures

- Now, to allocate an array of structures dynamically:

```
storeitem *ptr;
ptr = new storeitem[some_size];
```

- In this case, how would we access the first item?

```
ptr[0].item
```

Notice that the -> operator would be incorrect in this case because ptr[0] is not a pointer variable. Instead, it is simply an object. ptr is a pointer to the first element of an array of objects

Dynamic Structures

- What this tells us is that the -> operator expects a pointer variable as the first operand.
  - In this case, ptr[0] is not a pointer, but rather an instance of a structure. Just one of the elements of the array!
  - the . operator expects an object as the first operand...which is why it is used in this case!

Dynamic Structures

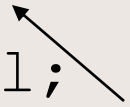
- Ok, what about passing pointers to functions?
- Pass by value and pass by value apply.
  - Passing a pointer by value makes a copy of the pointer variable (i.e., a copy of the address).
  - Passing a pointer by reference places an address of the pointer variable on the program stack.

Dynamic Structures

- Passing a pointer by value:

```
storeitem *ptr = new storeitem;
display(ptr);
```

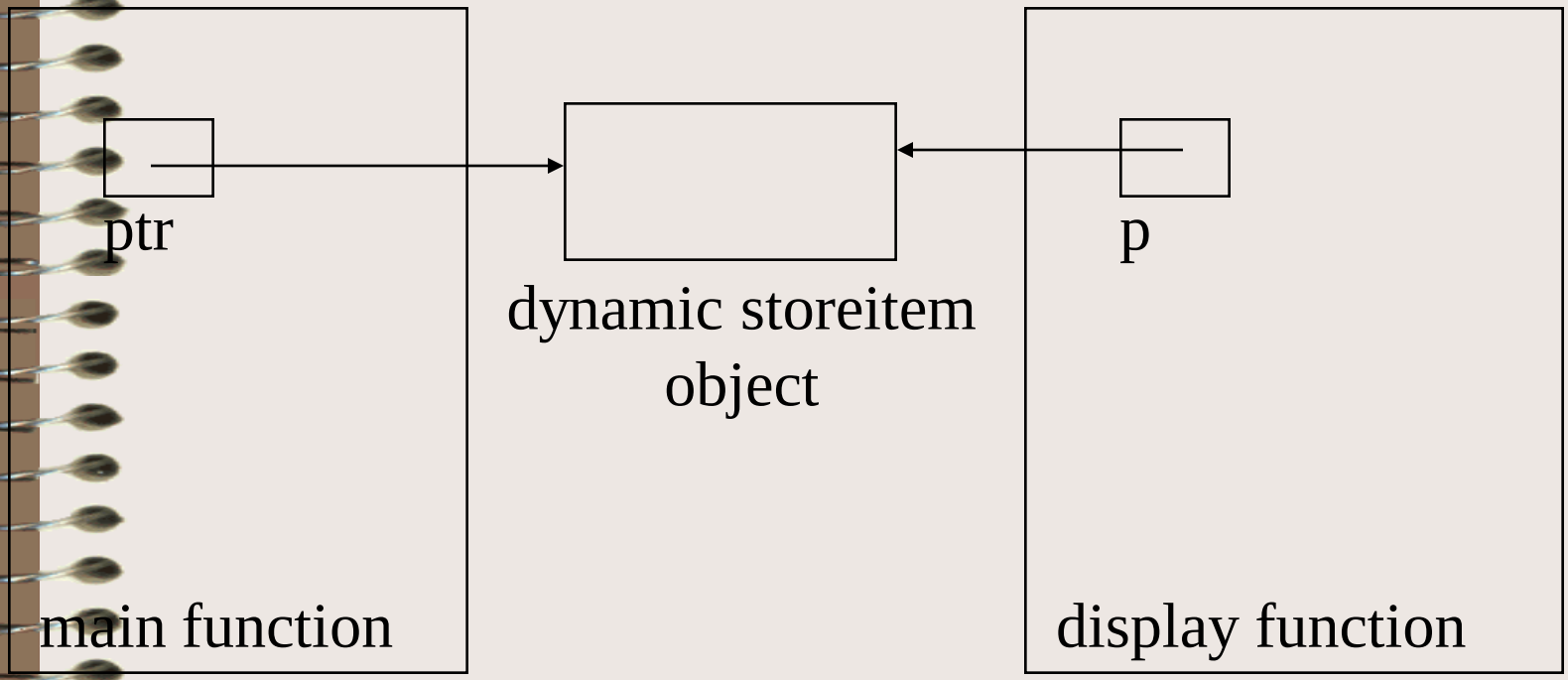
```
void display(storeitem * p) {
   cout <<p->item <<endl;
   }
```



p is a pointer to an object, passed by value. So, p is a local variable with an initial value of the address of a storeitem object

Dynamic Structures

- Here is the pointer diagram for the previous example:



Dynamic Structures

- Passing a pointer by reference allows us to modify the calling routine's pointer variable (not just the memory it references):

```
storeitem *ptr;  set(ptr);  cout <<ptr->item;
```

```
void set(storeitem * & p) {
  p = new storeitem;
  cin.get(p->item,100,'\n');
  cin.ignore(100,'\n');
}
```

The order of the * and & is critical!

Dynamic Structures

- But, what if we didn't want to waste memory for the item (100 characters may be way too big
- So, let's change our structure to include a dynamically allocated array:

```
struct storeitem {
   char * item;
   float cost;
   float price;
   int barcode;
   };
```

Dynamic Structures

- Rewriting the set function to take advantage of this:

```
storeitem *ptr;      set(ptr);
```

```
void set(storeitem * & p) {
  char temp[100];
  cin.get(temp,100,'\n');
  cin.ignore(100,'\n');
  p = new storeitem;
  p->item = new char[strlen(temp)+1];
  strcpy(p->item,temp); }
```

watch out for where the +1 is placed!