Kevin Chen

11/15/15

Compound Data Types

Arrays

* Declaration: type name [number of elements]; Example: int ages[5];
* Initializing: use curly braces: int age[5] = {1, 2, 3, 4, 5};
  + If fewer values than the array size are provided, then remaining values are set to default value (zero for fundamental types).
  + If you don’t specify size, the array will be the size of the number of values you initialize it to.
* Function declaration: omit the size in the first bracket, while specifying the size in the remaining brackets. Computer will ignore size in first bracket.
  + This is needed since only data is passed through; size of array is not.
  + Size of first dimension is lost when passing in array as parameter.
* Arrays cannot be assigned values. Only individual elements of arrays can be assigned values.
* As a parameter, only pointer is passed, not actual value of array.
* Library arrays: allow being copied (an expensive operation)
  + Declaration: array<int,3> myarray {10,20,30};
  + Defined under header <array> (must #include <array>)
  + Can access size of array with size().

Character Sequences

* string is an array of char. Null character ‘\0’ is added after last character in array of char to indicate end of sequence.
  + Can declare array of characters: char myword[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’} which creates an array of size 6.
  + Sequence of char enclosed in “” are literal constants. Their type is a null-terminated array of characters. Thus, will have ‘\0’ appended. Thus, can also declare char array to be a string: char myword[] = “Hello”; (Also size 6.)
* C-Strings (array of characters) vs. the class string.
  + The class string can change sizes and be set to values whereas C-strings have fixed sizes and cannot be set to other array values.
  + Use string with cin if you don’t know size of string that will be passed in.

Pointers

|  |  |  |
| --- | --- | --- |
|  | After a Type | In front of a variable |
| \* | Declaring pointer  int \*: pointer to int. | Dereference a pointer to get the value it is pointing to  \*p: dereference the pointer p |
| & | Declaring reference  int &: reference to int | Address-of operator  &p: access the address of value p |

* Address-of operator (&)
  + Use it in front of a variable to get the address of the first byte of memory it is located.
  + Example: int \* foo = &myvar; // Assuming myvar is an int.
* Reference – somewhere between a pointer and value.
  + Automatic conversion between value to reference when you pass a variable by reference or return a reference.
  + References can be used just like values syntactically, except you are modifying the original copy instead of a duplicate, local copy.
  + No automatic conversion between pointer to reference.
* Dereference operator (\*)
  + assigns to the value pointed by a variable.
  + Example: baz = \*foo; If foo had value 1776, baz would be assigned to value at memory address 1776.
* Declaration of pointers (also use \*): need to know data type the pointer is going to point to because a pointer has different properties depending on the type it is going to point to.
  + Syntax: type \* name; Example: int \* number; // number will point to an int.
  + The type is not of the type of the pointer itself, but instead the type of the value it is going to be point to.
* Can also access and change value at address if dereference operator is on left side. Example: {int \* foo; foo = (int \*) 1776; \*foo = 10;} Memory address at 1776 will have value of 10.
  + Note that if operator precedes pointer name, it is setting the value located at the address of the pointer. If operator does not precede it (the operator is not there), it is setting the value of the pointer to a new address.
* Pointer can be assigned to array variable without & because variable itself is an address of the first element of the array. Increment the pointer variable it to access the next element.
* Pointers can also be used like an array. So if “a” is a pointer, then you can do a[5] = 0 or instead \*(a + 5) = 0 to access 5 cells afterwards.
* Pointer initialization: [type] \* [pointer name] = [address] sets the pointer to the address.
  + Example: int \* myPointer = (int\*) 1776; sets the address contained in myPointer to 1776.
  + Note that dynamic allocation of memory (new Object\_Type) returns the address of the new object, so you can do Object\_Type \* name = new Object\_Type(parameters);
* Pointer arithmetic: only addition and subtraction
  + When you add 1 to address, you don’t actually add 1, but instead you add the size of the type. Example, if long is size 4, then {long \* mylong; ++mylong;} would add four to the address stored in mylong.
* Postfix (incrementing and decrementing) with dereferencing operators. Remember that postfix has higher precedence than dereference.
  + \*p++ // same as \*(p++): increment pointer, and dereference unincremented address
  + \*++p // same as \*(++p): increment pointer, and dereference incremented address
  + ++\*p // same as ++(\*p): dereference pointer, and increment the value it points to
  + (\*p)++ // dereference pointer, and post-increment the value it points to
  + Practice: What does this do? \*p++ = \*q++;
* If you just want to read memory, use constant during the declaration of pointers. Example (first one used more often): const int \* p = &y; -- OR – int const \* p = &y// Would be illegal to do \*p = 5;
  + Pointers to const cannot be implicitly converted to pointers to non-const, but pointers to non-const can be implictly converted to pointers to const.
  + Method that has parameter that points to non-const can modify value at the address. If parameter points to const, method cannot modify the value.
* Constant pointers: add const after asterisk, before pointer’s name. Can’t modify address that pointer is pointing to. Example: int \* const p = & y; // Would be illegal to do p = 1776;
* Pointers and string literals. Same rules apply as using arrays as pointers.
  + Example: const char \* foo = “hello”; // sets address that foo points to be the address location of “h”.
  + Can access the “o” in “hello” with foo[4] or \*(foo+4);
* Pointers to pointers (aka double pointers): Use multiple asterisks. Example: char \*\* c;
* Void pointers: points to values that has no type.
  + Flexibility: can point to any data type. Limitation: cannot directly dereference data pointed by the pointer.
  + Can be casted to a particular data type.
  + One use: pass generic parameters into function.
* Invalid pointers and null pointers
  + Pointers can point to any address, whether that address has an actual element or not. That’s okay. Error occurs when you actually dereference that value.
  + If pointer needs to point to nowhere, set its address to 0, nullptr, or NULL.
* Pointers to functions
  + Syntax: name of function is enclosed in parenthesis, asterisk inserted b4 name.
  + Example: int (\*minus)(int,int) = subtract; // subtract is the name of a method that takes in two parameters of type int.
  + Typical use of this: pass a function as an argument to another function.

Dynamic Memory

* Sometimes, memory needs of a program can only be determined during runtime. Thus, computers need to dynamically allocate memory
* Operators new and new[]
  + pointer = new [type] // allocates memory for one single element
  + pointer = new [type] [number\_of\_elements]
* No guarantees that memory allocation will be successful. Two ways to check if allocation is successful:
  + Exceptions: int \* foo = new int [5] // if allocation fails, exception is thrown.
  + “nothrow” object (defined in header “new”): foo = new (nothrow) int[5]; //If allocation failed, then foo is still a nullptr. (You can check if foo is a nullptr using if statement after trying to allocate memory.)
* Delete and delete[]
  + delete pointer; // deletes a single item.
  + delete pointer[]; // deletes an array of items.
* Dynamic memory in C
  + C language does not have new and delete. Instead has malloc, calloc, realloc and free, defined in the header <cstdlib> (known as <stdlib.h> in C)
  + Do not mix C++ functions with C functions.

Data Structures

* Group of data elements grouped under one name. A simple type of class.
* Syntax:

struct type\_name {

member\_type1 member\_name1;

member\_type2 member\_name2;

etc…

} object\_names;

* Access or set a member of a struct with the following syntax [struct name].[member name]
  + Struct type can be used as a parameter to a function.
  + Can create an array of structs instead of specifying names at the end.
* Pointers to structs: pointers point to the whole struct.
  + Use the “->” or the normal \* in order to dereference one of the members of the struct.
  + Example: pmovie is a pointer to a movie struct. Movie structs have titles:

pmovie->title –OR-- (\*pmovie).title.

* + This is NOT \*pmovie.title which is equivalent to \*(pmovie.title), which would access the value pointed by a hypothetical pointer in the movie struct.
* Nested structures get more fun.

Other Data Types

* Type Aliases: use a different name by which a type can be identified
  + Typedef: typedef existing\_type new\_type\_name ;

Example: typedef char C; // C myword; would create a char variable, myword.

* + Using: Example: using C = char;
  + Typedef has certain limitations when dealing with templates, while using does not. But typedef has been around longer.
  + Type aliases can be used as tools to abstract programs from the underlying types they use. For example, an alias of int that is used to store a particular kind of parameter can be all converted to long (or some other type) without changing all ints.
* Unions: allow one portion of memory to be accessed as different memory types. Syntax similar to structs, but totally different purpose:

union type\_name {

member\_type1 member\_name1;

member\_type2 member\_name2;

etc…

} object\_names;

* Example:

union mix\_t {

int l;

struct {

short hi;

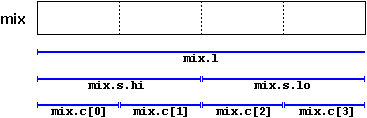
short lo;

} s;

char c[4];

} mix;

Potential distribution of memory. Changing mix.c[0] affects part of mix.s.hi.



* The exact alignment and order of the members of a union in memory depends on the system
* Anonymous unions: unions in class that are declared without a name. They are directly accessible from objects by: [object name].[member name]
* Enumerated types (enum):
  + Can only hold specific values. The possible values are called enumerators.

enum type\_name {

value1,

value2,

etc…

} object\_names;

* + All values enumerated types declared with “enum” are assigned an integer and can be implicitly converted to int. First value is 0, second value 1, etc. Can specify a different integer to begin with for the first value by assigning it to that value.
  + Can have enumerators that aren’t int or implicitly converted to int. Declare with “enum class” or “enum struct” instead of just “enum”. Each of enumerator values in enum class needs to be scoped into its type. (E.g. mycolor = Colors::blue;) This is only optional for enums.
  + Enum classes can also use any underlying type to determine the size of the type. Use colon and specify underlying type as shown in the example below:

enum class EyeColor : char {blue, green, brown};