Week 5 Lecture

# C Programming

## An Aside: Conditional Expressions

* Consider the following if-else statement.

|  |
| --- |
| if (x > y) /\* If x is greater than y, \*/  z = x; /\* then z equals x. \*/  else /\* Otherwise, \*/  z = y; /\* z equals y. \*/ |

* + There is an alternate way to write this using the ternary operator “?:”.

|  |
| --- |
| /\* Check if x > y. If it is, set z equal to y.  \*\* Otherwise, set z equal to y. \*/  z = (x > y) ? x : y; |

* The general form of this is .
  + is evaluated first.
  + If it is true (i.e. non-zero), then is evaluated and the result is taken as the value of the conditional expression.
  + Otherwise is evaluated and that result is taken as the value.
  + Note: Only one of and is evaluated.

## Pointers (continued)

### A recap from last week…

* A pointer variable points to an address in memory.
* A pointee is the memory location to which the pointer points.
* The code below shows how to declare a pointer and how to use the & and \* operators (K&R, p. 94).

|  |
| --- |
| int x = 1, y = 2, z[10];  int \*p; /\* p is a pointer to an int \*/  p = &x; /\* p now points to x \*/  y = \*p; /\* y is now 1 \*/  \*p = 0; /\* x is now 0 \*/  p = &z[0]; /\* p now points to z[0] \*/ |

* + The address of operator, &, returns a reference to a variable’s address.
  + The dereferencing operator, \*, returns the value stored at the address to which p points.
* **Three rules**:
  1. Pointer and pointee are separate, so don’t forget to set up the pointee.
  2. Dereference a pointer to access its pointee.
  3. Assignment (=) between pointers makes them point to the same pointee.

### Pointers and Arrays

* Any operation that can be achieved using arrays can also be achieved using pointers
  + Note that the pointer implementation of any operation will be faster in general
* The example that is on pages 98-99 of K&R, demonstrates the strong relationship between the two.
  + Because the name of an array is a synonym for the location of the first element in the array, the assignment can also be written as .
  + The two forms and are equivalent because in evaluating , C actually converts it to immediately.
    - So, this means that and are also equivalent ( is the address to the element beyond ).
  + One difference between an array name and a pointer is that a pointer is a variable (so and ++ are legal) but an array name isn’t a variable (so and ++ are not legal).

#### Arrays and Pointers as function parameters

* Even though an array name is not a variable, when an array name is passed to a function, the location to the initial element in the array is what is actually passed, so this argument is actually then a local variable while in that function (a pointer).
* As parameters in a function definition, and are equivalent.
  + is preferred since it explicitly says that the parameter is a pointer

#### Pointer Arithmetic

* If is a pointer to an element of an array, then ++ increments to point to the next element in the array (so increments it to point to i elements beyond where it previously pointed)

#### Character Pointers (C Strings)

* String constants are arrays of characters but can be represented using pointers.
* String is terminated with ‘\0’ (the null character) so the length in storage is the length of the string plus one
  + Don’t forget about that ‘\0’ or you will run into issues in your code.

## typedef

* A facility for creating new data type names.
* It makes the name a synonym for the type it defines, which can be used in declarations, casts, etc.

## Structures

* A **structure** is a user defined collection of variables (possibly of different types), which are grouped together for convenient handling.
  + The keyword **struct** introduces a structure declaration.
  + The **structure tag** (optional) names this structure and can be used as shorthand for the part of the declaration in braces.
  + The variables named in a structure are called **members**.
* A struct declaration defines a type.
  + The right brace that terminates the list of the member of that struct may be followed by a list of variables (just like for any basic type).
  + If it is not followed by a list of variables, no storage is reserved.
    - It just describes a template or the shape of a structure, in this case.
  + If a tagged, the tag can be used in later definitions of instances of the struct.
* **Example** (not using typedef but using tag):

|  |
| --- |
| struct point {  int x;  int y;  }  ...  struct point pt;  struct point maxpt = { 320, 200 }; |

* **Example** (using typedef and tag with different type name and tag name):

|  |
| --- |
| typedef struct point {  int x;  int y;  } coordinate;  ...  struct point pt1;  coordinate pt2; |

* **Example** (using typedef and tag with same type name and tag name):

|  |
| --- |
| typedef struct point {  int x;  int y;  } point;  ...  struct point pt1;  point pt2; |

* **Example** (using only typedef):

|  |
| --- |
| typedef struct {  int x;  int y;  } point;  ...  point pt1;  point pt2; |

* **Example** (using neither typedef nor tag)
  + **Note**: Only a good idea to do it this way if you need struct variables but do not plan to reuse the structure type.

|  |
| --- |
| struct {  int x;  int y;  } pt1, \*pt2, pt3[10];  ...  point pt1;  point pt2; |

* Accessing struct members
  + ‘.’ operator
    - used to access members of a struct or union
  + ‘->’ operator
    - use when you have a pointer to a struct and need to access its members
    - say you have struct point \*pt2
      * pt2->x equivalent to and a shortcut for (\*pt2).x
* **Note**: Structures cannot contain an instance of itself, but they can contain a pointer to an instance of itself.
  + **Example**: A list node to be used in a doubly linked list.

|  |
| --- |
| struct list\_node {  int data;  node \*prev;  node \*next;  }; |

## Unions

* Declaration looks like a struct definition, but unions behave a bit different.
  + Unions are really just structs in which all members have offset zero from the base, and the structure that is the union is big enough to hold the largest member.
    - Only one union member can be set at any given time.
      * Say you have a union with int x, float y, and char \*z members.
      * If you first write to y, but then you write to x and then to z, at the end, you will end up with the value you set it to be in z and “junk” in x and y.
        + So writing to x overwrote what we stored in y, and then writing to z overwrote what we stored in x.
  + The same operations are permitted on unions as on structures.

## Scope

* Scope of a name is the part of the program in which the name can be used
* Variables have scope

### Local Variables

* Variables that come into existence only when the unction they appear in is called and disappear when the function is exited. (i.e. they come and go with function invocation)
* So in the event that we have two functions (let’s call them and ) and a variable declared and set in each that have the same name and even type (let’s say it is ), they are unrelated to each other.
  + So can never access the variable that is local to , and can never access the variable that is local to .
* K&R refers to local variables as automatic variables.
* They do not retain their values from one function call to the next and must be explicitly set each time the function they occur in is invoked.

### External Variables

* Globally accessible variables
* Can be accessed by name by any function in the program
* Retain their values even after the function that set them has returned
* Useful as an alternative to argument lists to communicate data between functions
  + However, be careful when using these.
* Must be **defined** exactly once, outside of any function.
  + This sets aside the storage for it.
* Must also be **declared** in each function that wants to access it.
  + This states the type o the variable.
  + In some cases you may need to use the keyword extern at the beginning of these declarations but in most cases this is redundant.
    - The best practice is to place the definitions of all external variables at the top of the source file and omit the extern declarations entirely.
    - **Example**: A program that consists of several source files
      * Let , , and represent the source files of this program.
      * If an external variable is defined in and used in and , then extern declarations are needed in and to connect the occurrences of the variable

### Static Variables and Functions

* Static storage is specified by prefixing the normal declaration with the word .
* External static variables
  + The static declaration limits the scope of the variable being declared to the rest of the source file being compiled
  + Useful if you need to hide certain variables from calling functions in other files but need to share the variable within the file where it is declared
* Static functions
  + While most often used for variables, the external static declaration can also be applied to functions.
    - Normally, functions are global (i.e. visible to any part of the program).
    - If a function is declared static though, its name is not visible outside of the file in which it was declared.
* Internal static variables
  + Local to a particular function (just as automatic variables are) but (unlike automatics) they remain in existence rather than coming and going with each function call.
  + Provide private, permanent storage within a single function.

### Register Variables

* A register declaration advises the compiler that the variable that is being declared will be heavily used.
* The idea behind this is these variables should be placed in machine registers, which may result in smaller and faster programs, but compilers are free to ignore this.
* The register declaration can only be applied to automatic variables and formal parameters of a function.
* Register declarations are ignored for excess (i.e. more declared than can fit in registers) or disallowed declarations
* Note: It is not possible to take the address of a register variable, regardless of whether or not the variable is actually placed in a register.
* Additional restrictions may apply and are machine dependent.
* **Note**: Only use this type of declaration if you have a real need to do so.

### Related Note on Variable Initialization

* If not initialized explicitly, **external** and **static** variables are **guaranteed to be initialized to zero**.
* However, if not explicitly initialized, **automatic** and **register** variables **have undefined initial values**.