241 lecture note:

For week4(02)

• All information accessible to a running computer program must be stored somewhere in the computer's memory

* C provides the ability to access specific memory locations, using “pointers”.
* Memory locations are identified by their address.

• How long are the addresses?

Intel Core i7 has 64-bit addresses:

**int \*ip; // defines a variable of type integer pointer**

**printf(“%lu”, sizeof(ip));**

What is the output of the simple **printf** statement?

**Pointer basics**

• Address of the location containing the data

– all pointers are typed based on the type of entity that they point to;

– to declare a pointer, use \* preceding the variable name as in:

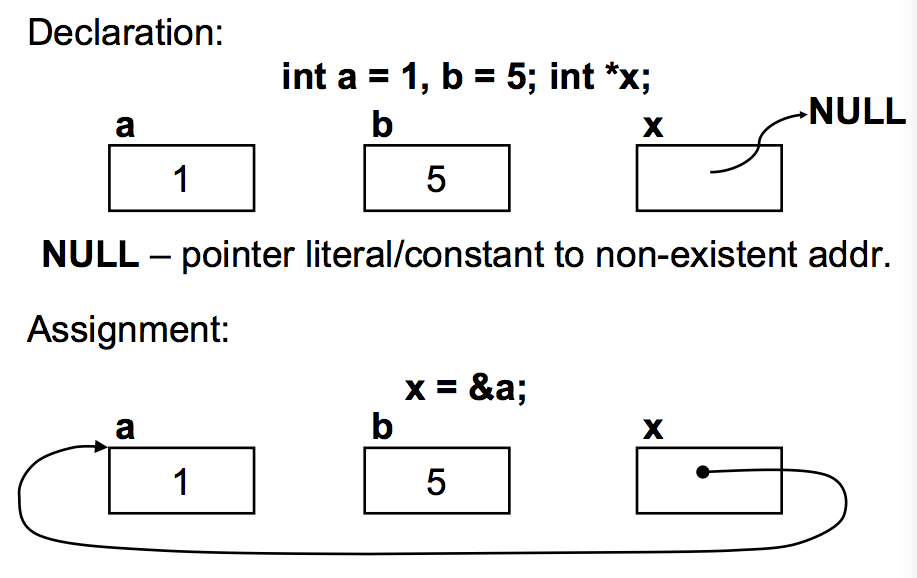
**int \*x;**

* To set a pointer to a variable’s address use & before the variable as in:

**x = &a;**

– & means “return the memory address of”

– x will now point to a, i.e., x stores a’s address



If you access x, you merely get the address

• To get the value in the variable/location that x points to, use \* as in

**\*x \*x = \*x + 1; // adds 1 to variable a whose**

**// address is contained in x**

• \* is known as the indirection (or dereferencing) operator as it requires a second access, that is, this is a form of indirect addressing. E.g.

**b = \*x;**

Recall:

int a = 1, b = 5;

int \*x;

x = &a; // What is the value of x ?

\*x = \*x + 1; // a = 2 ; b = 5 ;

b = \*x;

• What is the value of b ? 2

**Usage of pointers**

1. Provide an alternative means of accessing information stored in arrays, especially when working with strings; there exists an intimate link between arrays and pointers in C.
2. To handle variable parameters passed to functions.
3. To create dynamic data structures, that are built up from blocks of memory allocated from the heap at run time. This is only visible through the use of pointers.

**Pointers & Arrays**

Recall:

• Arrays in C are pointed to, i.e. the variable that you declare for the array is actually a pointer to the first array element

• You can interact with the array elements either through pointers or by using [ ]

-int z[], \*ip;

ip = &z[0];

z[0], \*ip or \*z can all be used to access the first element of the array z[]

What about accessing z[1] using pointers ?

\*(ip+1) or \*(z+1)

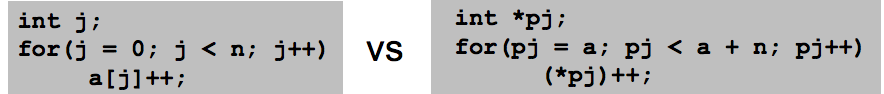
• Note that ip=ip+1 (or ip++) moves the pointer 4 bytes, instead of 1 to point to the next array element; amount added depends on size of array element

– 8 for an array of doubles

– 1 for an array of chars

– 4 for an array of ints

Iterating through elements of an array:



* pj is a pointer to an int

• Start with pj pointing at a, i.e., pj points to a[0]

• The loop iterates while pj < a + n

– pj is a pointer, so it is an address

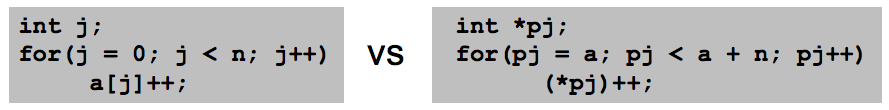
– a is a pointer to the beginning of an array of n elements; so a+n is the size of the array

– pj++ increments the pointer to point at the next element in the array

– The instruction (\*pj)++ says “take what pj points to and increment it”

**Pointers Arithmetic**

Iterating through elements of an array:



NOTE:

(\*pj)++; // increments what pj points to

\*(pj++); // increments the pointer to point at the // next array element

• What do each of these do?

\*pj++; // unary ++ acts on pj, before \*indirection

++\*pj;

For week5

**Variable Storage Class**

C storage classes are:

1. Auto
2. Static
3. Register
4. Extern

Storage class of a variable determines its:

• **Scope** attribute – where is a variable visible

• **Lifetime** attribute – how long does a variable exists

**Scope and Lifetime**

* Lifetime/storage attributes can be:

**– static** variables are allocated memory when program starts;

– **auto –** automatic variables are allocated memory when execution enters the block that contains it;

**– register** – reside in CPU’s high speed memory

• Scope attributes can be:

**– local** **– v** is only visible inside the current, innermost scope, independent of storage/lifetime attribute; e.g. there are local static variables in C

**– global – v** is visible in the whole compilation unit, from the line of declaration to the end of file

**– external – v** is visible in all compilation units; static

**auto Storage Class**

* **auto** is the default storage class for a variable defined inside a function body or a statement block

• **auto** prefix is optional; i.e. any locally declared variable is automatically **auto**, unless specifically defined to be static

Example:

**{**

**auto double x; /\* Same as: double x \*/**

**int num; /\* Same as: auto int num; \*/**

**. . .**

**}**

* *Automatic variables* may only be declared within functions and compound statements (blocks)

– Storage allocated when function or block is entered

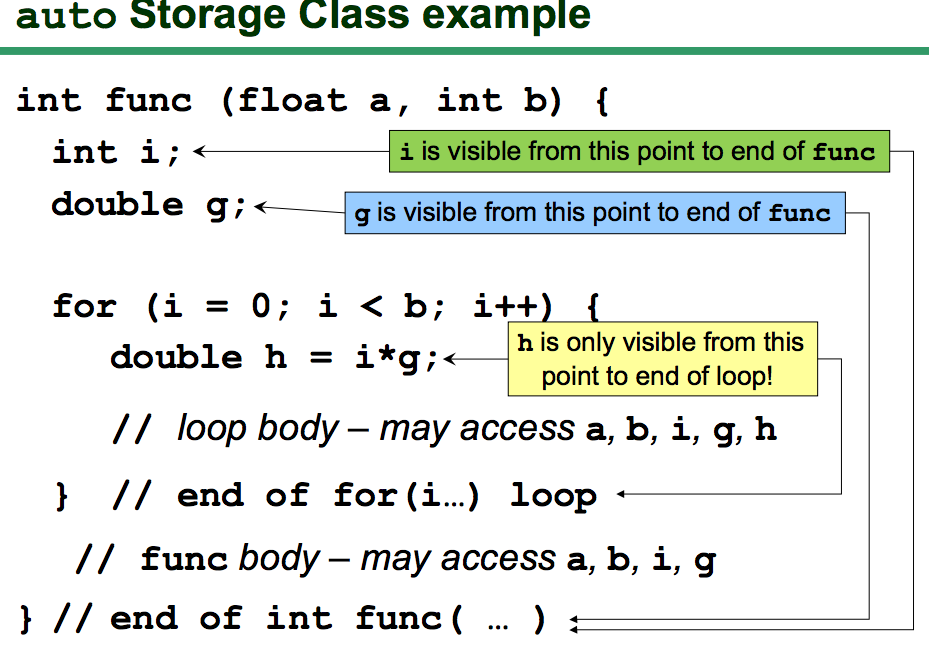
– Storage is released when function returns or block exits

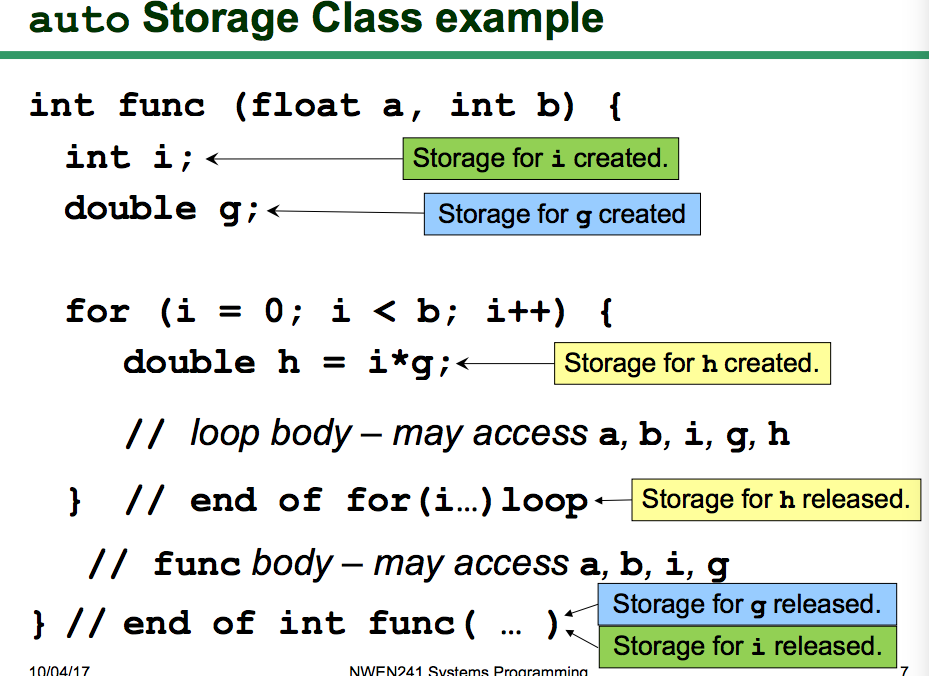
• Parameters and result are similar to automatic variables

– Storage is allocated and initialized by caller of function

– Storage is released after function returns to caller.

• Variables declared within a function or compound statement are visible only from the point of declaration to the end of that function or compound statement.





**auto Storage Class initialization**

If an **auto** variable is defined but not initialized:

– Variable has an unknown value when control enters its containing block

• If an **auto** variable is defined and initialized at the same time:

– Variable is re-initialized **each** time control enters its containing block

• An **auto** variable’s scope is limited to its containing block (i.e., it is **local** to the block)

**static Storage Class**

* Storage for a static variable:

– Is allocated when execution begins

– Exists for as long as the program is running

• A static variable may be defined either inside or outside a function’s body.

• The static prefix must be included

Example: static double seed

**static Storage Class initialization**

* If a static variable is defined but not initialized:

– Is set to zero (0) once, when storage is allocated

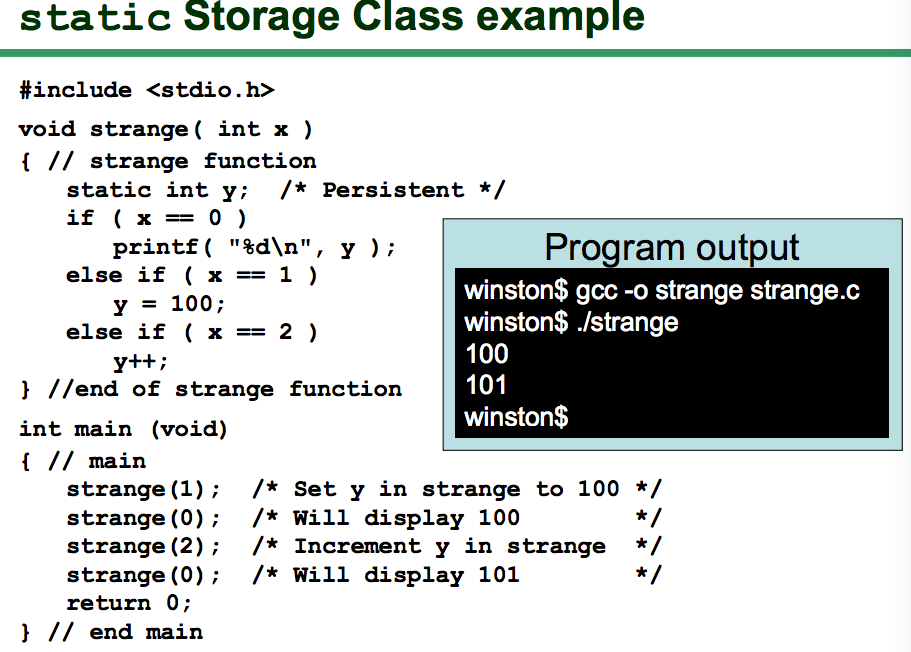
• If a static variable is simultaneously defined and initialized:

– Is initialized once, when storage is allocated

• A static variable defined inside a function body is visible only in its containing block

• A static variable defined outside a function body is visible to all blocks which follow it in the current compilation units

• If you wish it to be visible in other compilation units, it must be declared extern



**register Storage Class**

* The fastest storage resides within the CPU itself in high-speed memory cells called registers

• The programmer can request the compiler to use a CPU register for storage

Example:

register int k;

• The compiler can ignore the request, in which case the storage class defaults to auto

• Some machines, e.g. stack architectures, have no user visible register

**extern Storage Class (single source file)**

* **extern** is the default storage class for a variable defined outside a function’ s body

• Storage for an **extern** variable:

– Is allocated when execution begins

– Exists for as long as the program is running

• If an **extern** variable is defined but not initialized:

– Set to zero (0) once, when storage is allocated

• If an **extern** variable is defined and initialized:

– Initialized once, when storage is allocated

• An **extern** variable is visible in all functions that follow its definition (i.e., it is **global**)

**extern Storage Class example**

**#include <stdio.h>**

**float x = 1.5; /\* Definition - extern class - global \*/**

**void show (void) {**

**printf("%f\n", x); /\* Access global x \*/**

**}**

**int main (void) {**

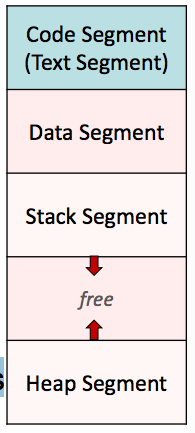
**printf("%f\n", x); /\* Access global x \*/**

**show();**

**return 0;**

**}**

**Storage Classes in Multiple Files**

1. Functions stored in a single source file can be divided into separate source files.
2. Variables defined in one source file can be accessed from other source files via the extern storage class.
3. An extern variable can be defined in one file only. However, it may be declared from other files.
4. An **extern** variable is defined exactly once in a file by placing it outside all blocks.
5. If an **extern** variable is not initialized at definition time → extern prefix must be omitted
6. If an **extern** variable is initialized at definition time → extern prefix is optional
7. An **extern** variable is declared in another file by using the **extern** prefix.

Example: **extern** int k;

**Memory Layout of a Program**

Memory space for program code includes space for machine language code and data

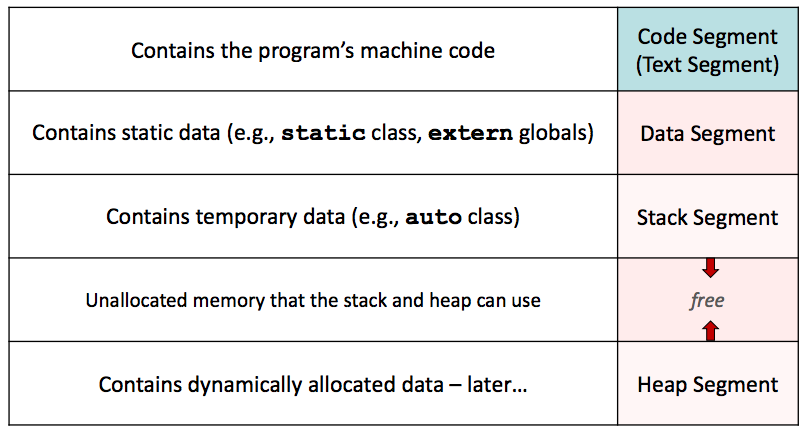
• Text / Code Segment

– Contains program’s machine code

• Data spread over:

1. Data Segment – Fixed space for global variables and constants
2. Stack Segment – For temporary data, e.g. local variables in a function; expands / shrinks as program runs
3. Heap Segment – For dynamically allocated memory; expands / shrinks as program runs

Where are auto, static, and extern variables stored?



For week6

* **Basic data types:** – int : integer ✓ – char : character ✓ – float : floating point number ✓ – double : double-precision floating point number ✓

• **Derived data types:** – Arrays ✓ – Strings ✓ – Structures and Unions

• **User defined data types** – New “types” including enumeration types

* **Derived types**

– Arrays – all elements must be of the same data type

– Strings – array of characters with null \0 character at end

• **What if you need a collection / group of information consisting of different data types?**

– E.g. student record that comprises name (last, first, middle and preferred), student ID, course, type, etc.

– Use a **composite structure** or record that is made up different basic/derived data types;

– Use a **composite union** if different types do not exist at the same time;

– Use **enumeration enum** to define list of constants

**Enumeration**

Enumeration is a user-defined data type. It is defined using the keyword enum and the syntax is:

enum tag\_name {name\_0, …, name\_n} ;

The tag\_name is not used directly. The names in the braces are symbolic constants that take on integer values from zero through n. As an example, the statement:

enum colors { red, yellow, green } ;

creates three constants. red is assigned the value 0, yellow is assigned 1 and green is assigned 2.

/\* This program uses enumerated data types to access the elements of an array \*/

**#include int main( ) {**

**int August[5][7] = {**

**{0,0,1,2,3,4,5},**

**{6,7,8,9,10,11,12},**

**{13,14,15,16,17,18,19},**

**{20,21,22,23,24,25,26},**

**{27,28,29,30,31,0,0}**

**};**

**enum days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};**

**enum week {week\_one, week\_two, week\_three, week\_four, week\_five};**

**printf ("Monday the third week of August "**

**"is August %d\n", August[week\_three][Mon]);**

**}**

**Structures**

A struct is a derived data type composed of members that are each fundamental or derived data types.

A single struct would store the data for one object. An array of structs would store the data for several objects.

A struct can be defined in several ways as illustrated in the following examples:

Declaring structure types

Syntax of the structure type:

struct struct\_type {

type1 id1;

type2 id2;

…

};

E.g.,

struct student\_info { // named struct

char name [20];

int student\_id;

int age;

}; // does not reserve any space

Declaring a variable **current\_student**

struct student\_info current\_student;

Above statement reserves space for:

– 20 character array,

– integer to store student ID, and

– integer to store age.

Declaring array of structures to store information of enrolled students in a class

struct student\_info nwen241class[250];

Reserves space for 250 element array of records (structs) for students enrolled in NWEN241.

**Creating new user defined types**

• Instead of saying struct student\_info every time we declare a variable, we can define it as a new data type, e.g.

typedef struct { // unamed struct

char name [20];

int student\_id;

int age;

} StudentInfo;

* This makes StudentInfo a new user-defined type, and you can declare a variable as follows:

StudentInfo current\_student;

New struct and data type

* If student\_info has been previously defined, then we can create a new data type using typedef :

typedef struct student\_info StudentInfo;

Or, we can also do this:

typedef struct student\_info {

char name [20];

int student\_id;

int age;

} StudentInfo;

**Accessing and manipulating structs**

We can reference a component of a structure by the direct component selection operator, which is a period, e.g.

strcpy(student1.name, “John Smith”);

student1.age = 18;

printf(“%s is in age %d\n”, student1.name, student1.age);

• The direct component selection operator has the highest priority in the operator precedence.

* The copy of an entire structure can be easily done by the assignment operator.

student1 = student2;

**Example – struct and typedef (1)**

**#include <stdio.h>**

**#include <String.h>**

**int main() {**

**typedef struct student\_info {**

**char name[20];**

**int student\_id;**

**int age;**

**} StudentInfo;**

**StudentInfo current\_student; // declare new variable using // new type StudentInfo**

**struct student\_info new\_student; // declare using struct // format**

**// do stuff – see next slide**

**}**

**Example – struct and typedef (2)**

**#include <stdio.h>**

**#include <String.h>**

**int main() {**

**// declarations in previous slide**

**…**

**// create new student record**

**strcpy(new\_student.name , "John Smith");**

**new\_student.student\_id = 300300300;**

**new\_student.age = 22;**

**current\_student = new\_student;**

**printf("Student name : %s\n", current\_student.name);**

**printf("Student ID : %.9d\n", current\_student.student\_id);**

**printf("Student Age : %d\n", current\_student.age); }**

**}**

**struct as function input parameter (1)**

• Suppose there is a structure defined as follows.

**typedef struct {**

**char name[20];**

**double diameter;**

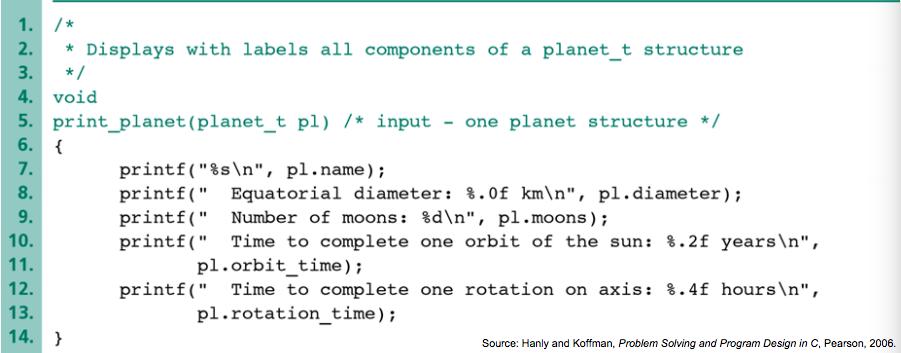
**int moons;**

**double orbit\_time, rotation\_time;**

**} planet\_t;**

**struct as function input parameter (2)**

* When a structure variable is passed as an input argument to a function, all its component values are copied into the local structure variable.



**struct as function input/output parameter (2)**

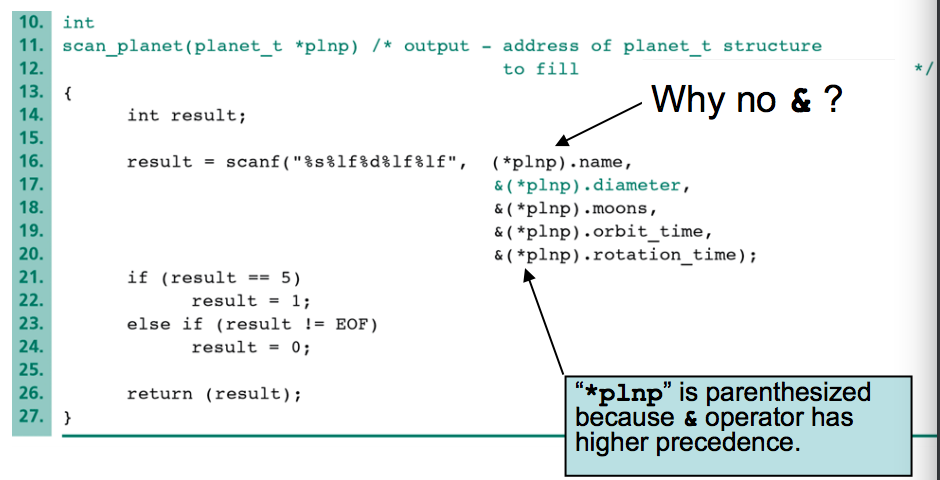
* If we define a variable as follows to store data to be read in:

**planet\_t current\_planet;**

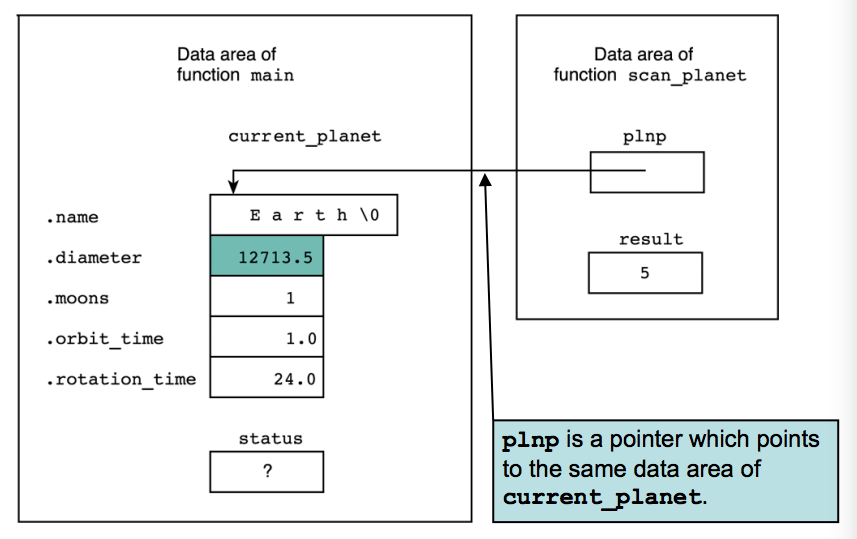
• For the following function, we call it by passing the parameter by reference:

**scan\_planet(&current\_planet);**

where the input argument is also used to store the result.

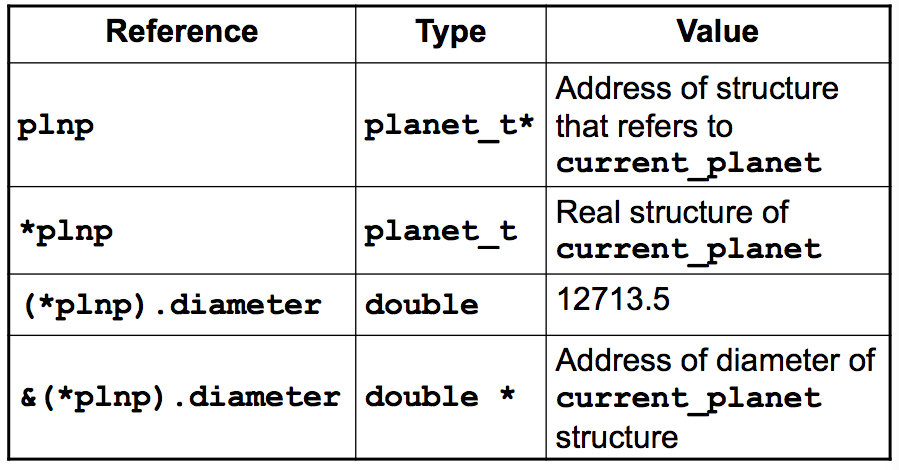


**Data Areas of function call**



**Indirect referencing steps**

• &(\*plnp).diameter is evaluated as shown in the following:



* In the above example, we use direct component selection operator: period, e.g.,

**&(\*plnp).diameter**

• C also provides indirect component selection operator : -> , e.g.

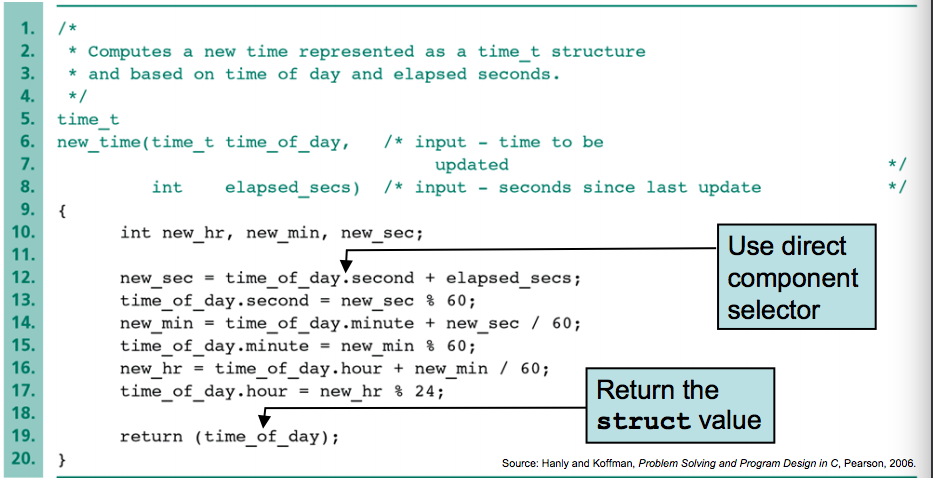
**&plnp->diameter**

is the same as

**&(\*plnp).diameter**

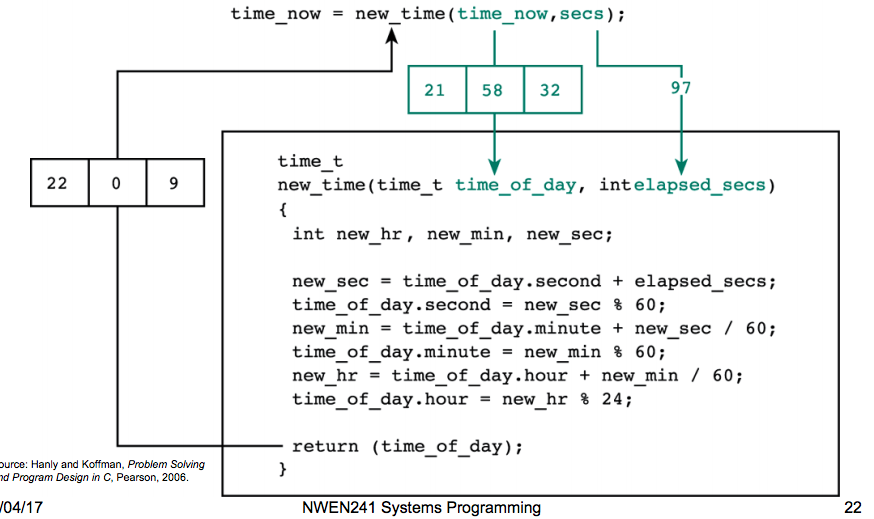
**Function returning a struct result type**

• struct variable can also be used as a return value of a function

****

**Function returning a struct result type e.g**.

Suppose the current time is 21:58:32, and the elapsed time is 97 seconds.



**Array of Structures (1)**

* An array of structures can be defined as follows:

**typedef struct {**

**int student\_id;**

**double gpa;**

**} student\_t;**

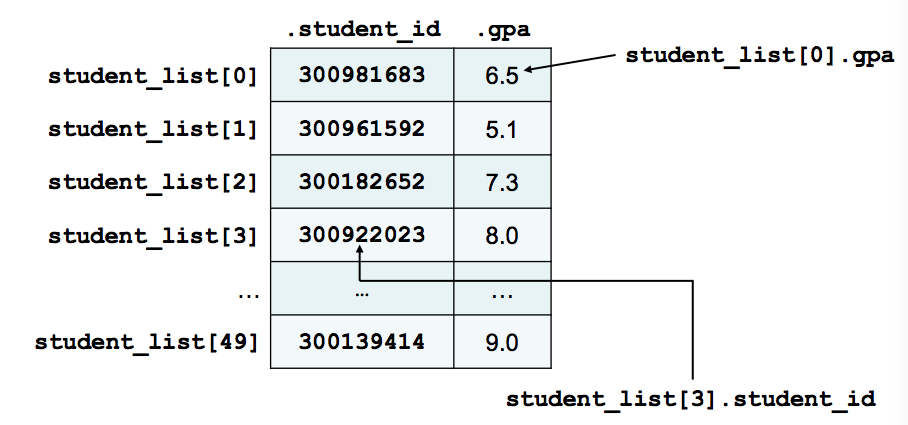
**student\_t student\_list[50];**

**student\_list[3].student\_id = 300922023;**

**student\_list[3].gpa = 8.0;**

**Array of Structures (2)**

* Can be simply manipulated as arrays of simple data types

****

**Unions**

• A union is like a struct, but the different fields take up the same space within memory

union space {

int i;

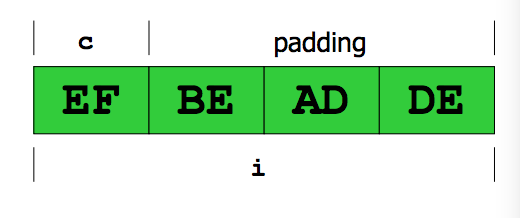
float f;

char c[4];

};

* sizeof(union space) = max ( sizeof(i), sizeof(f), sizeof(c))

**union example**

****

union AnElt {

int i;

char c;

} elt1, elt2;

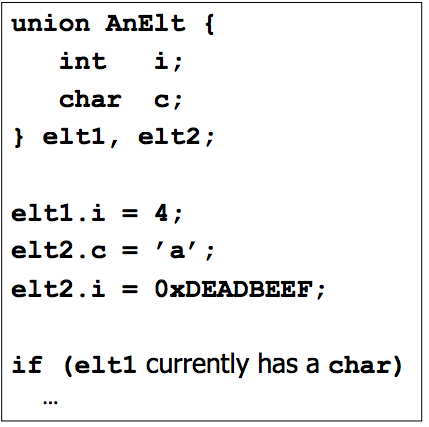
elt1.i = 4;

elt2.c = ‘ a‘

elt2.i = 0xDEADBEEF;

**union doesn’t know what it contains…**

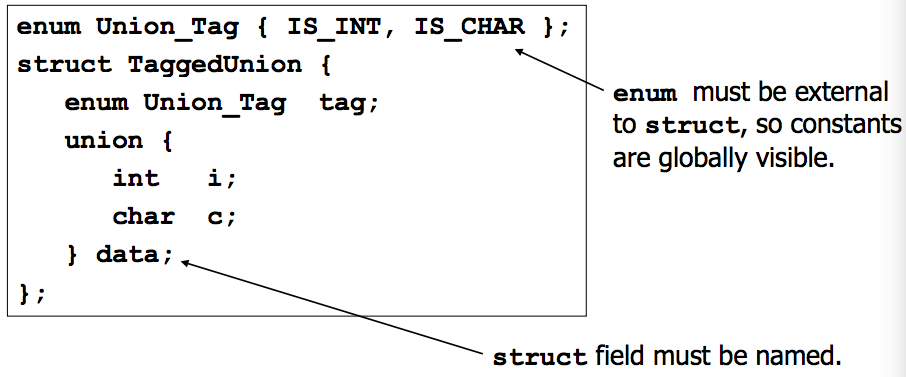
* How should your program keep track whether elt1, elt2 hold an int or a char?

****

• Basic answer: Another variable holds that info

**Tagged** unions

* Tag every value with its case
* Pair the type info together with the union – implicit in other programming languages like Java.



For week7

Command line arguments are parameters supplied to a program, when the program is invoked.

main can take 2 arguments, conventionally called argc and argv.

int main(int argc, char\* argv[])

argc

– Number of arguments (including program name)

argv

– Array of char\*s (that is, an array of ‘c’ strings)

– argv[0] à program name

– argv[1] à first argument

– …

– argv[argc-1] à last argument

$ ./main\_arg NWEN241 is about Systems Programming using C 8 arguments

0: ./main\_arg

1: NWEN241

2: is

3: about

4: Systems

5: Programming

6: using

7: C

$

Total of 8 arguments including program name itself. Arguments are read in as strings.

In general, I/O is the process of copying data between main memory and external devices

In C, everything is a file; --🡪each file is simply a sequential stream of bytes;

C imposes no structure on a file.

BUT, Defined in stdio.h is the struct FILE that comprises a file descriptor and a file control block

A file must first be opened properly before it can be accessed for reading or writing.

When a file is opened, a stream is associated with the file. Pointer to (i.e. address of) the “file” is returned

Input / Output & stdio.h

Every UNIX/Linux process begins with three open files corresponding to the standard input, output and error streams, macros defined in stdio.h:

****

Also defined in stdio.h are three variable types (including FILE), several macros (including above) and various functions for performing input / output

e.g. printf(), scanf(), getchar() , gets(), putchar(), puts(), etc.

File operations

1. Creating a new file
2. Opening an existing file
3. Writing data to a file
4. Reading data from a file
5. Closing a file
6. Random access operations

Declaring FILE pointer and Opening file

A file must be “opened” before it can be used.

FILE \*fp; // pointer to data type FILE

fp = fopen (filename, mode);

fopen 🡪returns a pointer (fp) to the file;

* used in all subsequent file operations.

mode 🡪 “r” – open the file for reading only

“w” – open the file for writing only

“a” – open the file for appending data to it

Did the fopen(…) command succeed?

If the file was not able to be opened, then the value returned by the fopen routine is NULL.

For example, if the file mydata does not exist, then:

**FILE \*fptr ;**

**fptr = fopen ("mydata", "r") ;**

**if (fptr == NULL) {**

**printf ("File open failed.\n");**

**}**

Closing a file

After completing all operations on a file, it must be closed to ensure that all file data stored in memory buffers are written to the file.

**General format:** fclose (file\_pointer);

FILE \*fp; // pointer to data type FILE

:::

fp = fopen (filename, mode);

:::

fclose (fp); // close the file

Read/Write Operations on Files

Simplest file input-output (I/O) function: **getc & putc**

**char ch;**

**FILE \*fp;**

**:::**

**ch = getc(fp);**

**getc** will return an end-of-file marker EOF, when the

end of the file has been reached.

**putc** is used to write a character to a file.

**char ch;**

**FILE \*fp;**

**:::**

**putc(c, fp);**

**main() {**

**FILE \*ifp, \*ofp;**

**char c;**

**ifp = fopen ("ifile.dat" , " r ");**

**ofp = fopen ("ofile.dat" , " w ");**

**while ((c = getc (ifp)) != EOF)**

**putc (toupper(c), ofp);**

**fclose (ifp);**

**fclose (ofp);**

**}**

1. fgetc() vs getc()

• **fgetc** is a subroutine that performs the same function as the **getc** macro; **fgetc** is NOT a macro.

• **fgetc** subroutine runs more slowly than **getc** but takes less disk space.

• Benefit: fgetc(\*p++) works but getc(\*p++) fails

1. fputc() vs putc()

• fputc is a subroutine while putc is a macro;

• same considerations for fputc as fgetc.

fscanf()

Same as scanf except need to **file pointer** as an argument.

Example:

int a, b;

FILE \*fptr1;

fptr1 = fopen ("datafile", "r");

fscanf( fptr1, "%d%d", &a, &b);

fscanf would read values from the file "pointed"

to by **fptr1** and **assign those values** to **a** and **b**.

End of File using EOF

The end-of-file indicator **EOF** informs the program when there are no more data (no more bytes) to be processed.

Check the value returned by the **fscanf** function:

**int istatus, var;**

**istatus = fscanf (fptr1, "%d", &var) ;**

**if ( istatus == EOF ) {**

**printf ("End-of-file encountered.\n") ;**

**}**

End of File using feof()

Use the **feof** function which returns a true or false condition:

**fscanf (fptr1, "%d", &var) ;**

**if ( feof (fptr1) ) {**

**printf ("End-of-file encountered.\n");**

**}**

**fprinf()**

Same as **printf** except need to file pointer as an argument.

**int a=5, b=20;**

**FILE \*fptr2;**

**fptr1 = fopen ("results", "w");**

**fprintf (fptr2, "%d %d\n", a, b);**

**fprintf** functions would write the values stored in **a and b** to the file "pointed" to by fptr2.

**Example using fscanf() & fprintf()**

#include

int main ( ) {

FILE \*outfile, \*infile ;

int b = 5, f ;

float a = 13.72, c = 6.68, e, g ;

outfile = fopen ("testdata", "w") ;

fprintf (outfile, "%6.2f%2d%5.2f", a, b, c) ;

fclose (outfile) ;

infile = fopen ("testdata", "r") ;

fscanf (infile,"%f %d %f", &e, &f, &g) ;

printf ("%6.2f,%2d,%5.2f\n", e, f, g) ;

fclose (outfile) ;

}

**Handling binary files**

Same as dealing with text files except in the opening step.

Need to open the file as a binary file using the binary mode identifier,

e.g.

**– "rb" r for read and b for binary**

**– "wb" w for write and b for binary**

**– ”ab" a for append and b for binary**

Example:

**FILE \*ptr;**

**ptr = fopen ("file1.exe","rb");**

**Reading binary files**

**fread** reads a block of binary data, up to **nmemb** elements of size,

**size** from **stream**, storing them at the address specified by **ptr.**

**size\_t fread ( void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);**

**fread** returns the actual number of elements read.

Example:

**unsigned char buffer[10];**

**FILE \*ptr;**

**ptr = fopen("file1.exe","rb");**

**fread (buffer, sizeof(buffer), 1, ptr);**

Writing binary files

**fwrite** writes a block of binary data comprising **nmemb** elements of size,

**size** from **ptr** to **stream**.

**size\_t fwrite (const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);**

**fwrite** returns the number of elements written.

Example:

**unsigned char buffer[10];**

**FILE \*write\_ptr;**

**write\_ptr = fopen("file2.exe","wb");**

**fwrite (buffer,sizeof(buffer),1,write\_ptr);**

Random Access (1)

Most often used with binary files using **fseek, ftell and rewind**.

**fseek** allows repositioning within a file.

**int fseek(FILE \*stream, long int offset, int startpoint);**

New position in the file is determined by:

**offset** – byte count (possibly -ve) relative to the position specified

by **startpoint** where

**startpoint = {SEEK\_SET, SEEK\_CUR, SEEK\_END}**

**| | |**

**Beginning of file Current file position End of file**

**Random Access (2)**

**ftell** returns the current file position:

**long int ftell(FILE \*stream);**

This may be saved and later passed to **fseek:**

**long int file\_pos;**

**file\_pos = ftell(fp);**

**…**

**fseek(fp, file\_pos, SEEK\_SET);**

/\* return to previous position \*/

**rewind(fp)** is equivalent to:

**fseek(fp, 0, SEEK\_SET).**