241 lecture note:

For week3

**Arrays in General**

• An array is a collection of data that holds a **fixed** number of data (values) of the **same type**

• In C, arrays and pointers are closely related concepts

– An array name by itself is treated as a constant pointer

----------------------------------------------------------------------------------------------------------

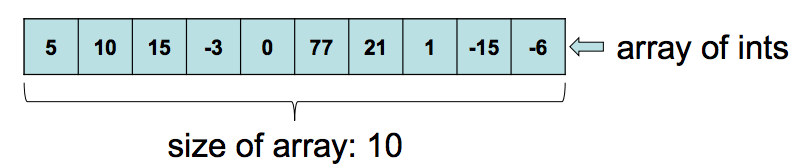
• We distinguish between two types of arrays:

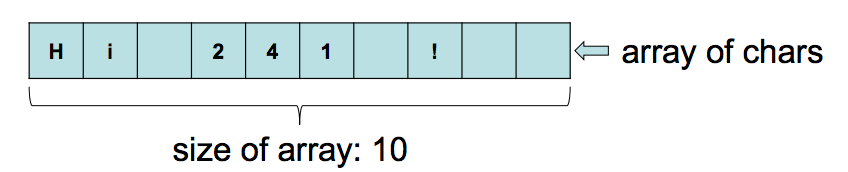
**– One-dimensional arrays**

**– Multi-dimensional arrays**

* The C language places no limits on the number of dimensions in an array, though specific implementations may

**One-dimensional (1D) Array Overview**







**Arrays**

• The simplest interpretation of an array is one-dimensional array, often referred to as a list

• The individual elements of the array can be accessed via indices



**Declaring 1D Arrays**

• Syntax for declaring an array: data\_type array\_name[array\_size];

• Example:

– We declare an array named data of floating-point type and size 4 as:

float data[4];

– It can hold 4 floating-point values

• The size and type of arrays cannot be changed after their declaration!

**Initializing 1D Arrays – Method 1**

• Arrays can be initialized **one-by-one**

• For example:

float data[4];

data[0] = 22.5;

data[1] = 23.1;

data[2] = 23.7;

data[3] = 24.8;

• In the case of large arrays this method is inefficient

**Initializing 1D Arrays – Method 2**

• Arrays can be also initialized when they are declared (just as any other variables):

float data[4] = {22.5, 23.1, 23.7, 24.8};

• An array may be partially initialized, by providing fewer data items than the size of the array

float data[4] = {22.5, 23.1};

– The remaining array elements will be automatically initialized to zero

• If an array is to be completely initialized, the dimension (size) of the array is not required

float data[] = {22.5, 23.1, 23.7, 24.8}

– The compiler will automatically size the array to fit the initialized data

Arrays & Loops

• Arrays are commonly used in conjunction with loops

– in order to perform the same calculations on all (or some part) of the data items in the array:

--While loop:

int idx = 0;

while(idx < 10){

// magic happens here

}

– For loop:

for (int idx = 0; idx < 10; idx++){

// magic happens here

}

**Off-by-one Error**

• The most common mistake when working with arrays in C is forgetting that indices start at zero and stop one less than the array size

* We often refer to this issue as ”Off-by-one Error”

int data[] = {1,2,3,4,5};//number of elements is 5

for (int idx = 0; idx <= 5; idx++){

// magic happens here

}

• The compiler does not control the limits of the array!

• This type of error can be detected using static code analysis

– For example using the **cppcheck** tool

• A Special type of Off-by-one error is the “Fencepost error”

– A straight fence with n sections has n+1 posts

**Determining the size of an array**

* The size of an array can be determined using the **sizeof()** operator

• It will **return the number of bytes the array "occupies" in the memory**

int data[] = {1,2,3,4,5};

printf("sizeof(data): %ld\n",sizeof(data)/sizeof(int));

• To determine the number of elements in the array, the returned value must be divided by the number of bytes reserved for the data type !

**Passing 1D Arrays to Functions (1)**

• Passing a single array element to a function

– can be passed in a similar manner as passing a variable to a function

void display(int age) {

printf("%d", age);

}

order!!!

int main() {

int age[] = { 18, 19, 20 };

display(age[2]); //Passing array element age[2] only

return 0;

}

**Passing 1D Arrays to Functions (2)**

Passing an entire array to a function

– When passing an array as an argument to a function, it is passed by its memory address (starting address of the memory area) and not its value!

float average(int age[]) {

int sum = 0;

for (int i = 0; i < 6; ++i) {

sum += age[i];

}

float avg = ((float)sum / 6);

return avg;

}

int main() {

int age[] = {18,19,20,21,22,23};

float avg = average(age);

printf("Average age=%.2f\n", avg);

}

**Searching Algorithms**

A search algorithm is an algorithm that retrieves information stored in some data structure

– Data structures can include linked lists, arrays, hash tables, etc.

• In C programming searching through the data of an array is a common operation

– Searching for a value in a large sized array is a resource and time demanding task

– Search functions are usually evaluated on the basis of their complexity, or maximum theoretical run time.

– The appropriate search algorithm often depends on the data structure being searched

------------------ Linear vs Binary search -----------------------------------

**Sorting Algorithms**

* A sorting algorithm is an algorithm that puts elements of a list in a certain order

• The most-used orders are numerical order and lexicographical order

• Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists

• Since lists allow only sequential access, the data is often taken to be in an array

**Bubble Sort**

Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one).

After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

**Unsorted: 4 1 5 9 8 2 3**

**Step 1 : 1 4 5 8 2 3 9**

**Step 2 : 1 4 5 2 3 8 9**

**Step 3 : 1 4 2 3 5 8 9**

**Step 4 : 1 2 3 4 5 8 9**

**stdlib.h**

In practice you do not have to know how to implement these algorithms

– The most efficient ones have been already implemented in C

– In the **stdlib.h** pre-processor directive (header file)

**• Binary search**

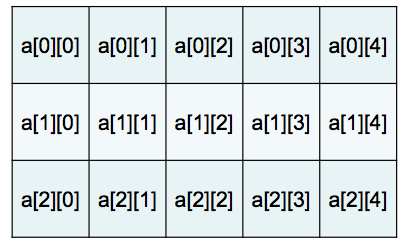
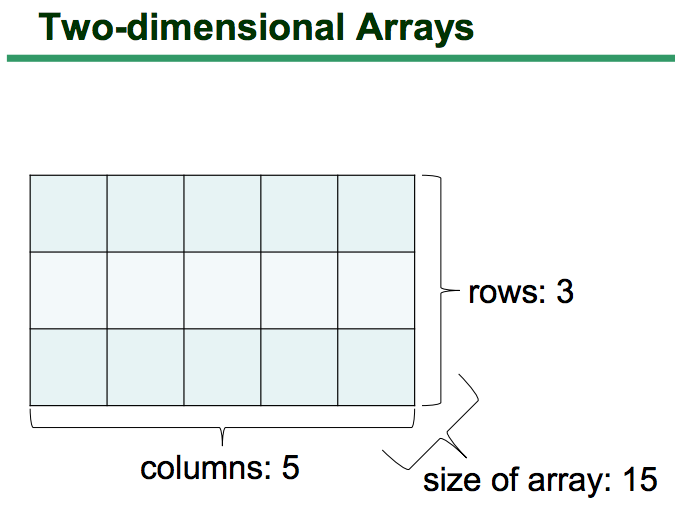
**• Quick sort**

**Multi-dimensional Arrays**

* In C, you can create array of an array known as multi-dimensional array

• The simplest interpretation of a multi-dimensional array is a table, i.e. a **two-dimensional array**

– each row has the same number of columns



**2D Arrays**

Declaring a char array with 3 rows and 5 columns

char two\_d[3][5];

– The array can hold 15 elements

• Accessing a value

char ch;

ch = two\_d[2][4];

• Modifying a value

two\_d[0][0] = 'x';

• The array can be initialized in one of the following ways

int two\_d[2][3] = {{ 5, 2, 1 }, { 6, 7, 8 }};

int two\_d[2][3] = { 5, 2, 1 , 6, 7, 8 };

int two\_d[][3] = {{ 5, 2, 1 }, { 6, 7, 8 }};

– **The number of columns must be explicitly stated.** The compiler will find the appropriate amount of rows based on the initializer list.

**Passing 2D Arrays to Functions**

* Similarly to one-dimensional arrays, a two-dimensional array element or an entire two-dimensional array can be passed to a function.

• When passing a multi-dimensional array as an argument to a function, the array is passed to the function by its memory address (starting address of the memory area) and not its value!

void enterData(int firstMatrix[][10], int secondMatrix[][10]) {

// code for reading and saving data into the 2D array

}

int main() {

int firstMatrix[10][10], secondMatrix[10][10];

enterData(firstMatrix, secondMatrix); // Calling Function to take data

}

**Three-dimensional Arrays**

* Declaring a three-dimensional (3d) array

float three\_d[2][4][3];

– Here, three\_d can hold 24 elements. Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements.

• Initializing a three-dimensional array

int test[2][3][4]={{{3, 4, 2, 3},{0, -3, 9, 11},{23, 12, 23, 2}}, {{13, 4, 56, 3},{5, 9, 3, 5},{3, 1, 4, 9}} };

For week4(01)

**Strings in General**

* C language does not support strings as a data type

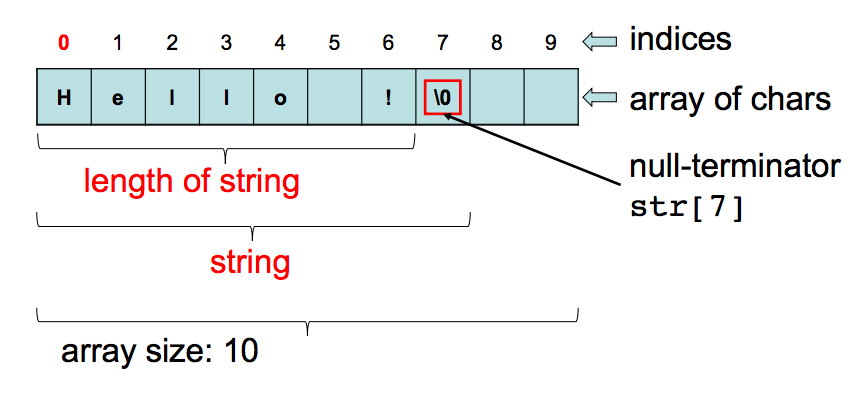
• A string is a sequence of characters that is treated as a single data item and terminated by a null character also known as the null-terminator, null byte or just '\0'

– In C language a string is actually a one-dimensional array of characters

* In C we distinguish between **String** **Literals** and **String** **Variables**

– If we (declare and) initialize a string using a pointer, we speak about a **String Literal** – If we (declare and) initialize a string using an array of characters, we speak about a **String** **Variable**

**Strings in General (cont.)**



**String Literals**

* String Literal, also known as a string constant or constant string, is a string of characters enclosed in double quotes, e.g.:

"NWEN241-2017 is driving me crazy!”

• String Literals are stored in C as an array of chars, terminated by '\0'

– '\0' is not the same as the '0' character, the integer 0, the double 0.0, or the pointer NULL

• String literals may contain as few as one or even zero characters

– Do not confuse a single-character string literal, e.g. "A" with a character constant, 'A'

• The former is actually two characters, because of the null-terminator stored at the end

– An empty string, "", consists of **only the null-terminato**r, and is considered to have a string length of zero, because the null-terminator does not count when determining string lengths

– String literals may contain any valid characters, including escape sequences such as \n, \t, etc.

**Passing String Literals to Functions**

* String literals are passed to functions as pointers to a stored string. For example, given the statement:

printf("Hello World!");

* The string literal "Hello World!" will be stored somewhere in memory, and the address will be passed to printf()

• The first argument to printf() is actually defined as a char \*

**Continuing a String Literal**

* If a string literal needs to be continued across multiple lines, there are three options:

printf("This will print over three lines, (and will include extra tabs or spaces)");

printf("This will \ print over a single \ line, (but will still include extra tabs or spaces)"); printf("This will " "print over a single " "line, (without extra tabs or spaces)");

**Operations on String Literals**

* Character pointers may hold the address of string literals

• String literals may be subscripted

char \*str = "Hello"; // Character pointers

printf("%c\n", str[0]); // will print 'H’

printf("%c\n", str[2]); // will print 'l’

printf("%c\n", str[5]); // will print "nothing". //Actually this will print the null-terminator

* Attempting to modify a string literal is undefined, and may cause problems in different ways depending on the compiler, e.g.

str[0] = 'S'; //NEVER do this when you declared and //initialized your string using pointers!

**String Variables**

* String Variables are typically stored as arrays of chars, terminated by a null-terminator

• String variables can be initialized either

– with individual characters (you have to supply the ‘\0’ explicitly)

– or more commonly and easily with string literals

as: array size; ss: string size;

char str[ ] = {'H','e','l','l','o','\0'}; // as=6, ss=5

char str[6] = {'H','e','l','l','o','\0'}; // as=6, ss=5

-------------------------------------------------------------------

char str[5] = "Hello"; //5 chars, sacrifices null-term.

char str[ ] = "Hello"; // as=6, ss=5

char str[20] = "Hello"; // ss=5, 15 null-term.

------------------------------------------------------------------

char str[ ];

str = "Hello"; // illegal method

char str[8];

str = "Hello World!"; // illegal method

char str[3] = "Hello"; // illegal method

**Character Arrays vs Character Pointers**

char s6[ ] = "hello";

char \*s7 = "hello";

• s6 is a fixed constant address, determined by the compiler

• s6 allocates space for exactly 6 bytes

• the contents of s6, can be changed, e.g. s6[0] = 'J’;

• s7 is a pointer variable, that can be changed to point elsewhere

• s7 allocates space for 10 (typically) - 6 for the characters plus another 4 for the pointer variable.

• the contents of s7 should not be changed

**The null-terminator**

• Any array of characters that ends with a '\0' is a string

• What comes after the end of the string doesn't matter, since the string has ended

char str[] = "One\0Two";

printf("%s\n", str);

• The program will print only the string "One”

– The '\0' character terminates the string

– What comes after, does not matter

• The array will contain 8 elements

– The string "One\0Two”, and

– another null-terminator, which was put at the end by the compiler

**Displaying Strings – printf()**

• Strings can be displayed on the screen using printf(); printf("%s\n", str);

* The precision ('%.N') parameter limits the length of longer strings

printf( "%.5s\n" , "abcdefg" ); // only ”abcde” will be displayed

• The width ('%N') parameter can be used to print a short string in a long space

printf( "%5s\n" , "abc" ); // prints " abc". Note // the leading two spaced at the beginning.

**Displaying Strings – puts()**

• The puts() function writes the string out to standard output and automatically appends a newline character at the end

char str[] = "This is an ";

printf("%s", str);

puts("example string.");

printf("See??\n");

• The output will be:

This is an example string.

See??

**Reading in Strings – scanf()**

• The standard format specifier for reading strings with scanf() is **%s** that the '**&**' is not required in the case of strings, since the string is a memory address itself

• scanf() appends a ‘\0’ to the end of the character string stored

• scanf() does skip over any leading whitespace characters in order to find the first non-whitespace character

• The width field can be used to limit the maximum number of characters to read from the input

• You should use one character less as input than the size of the array used for holding the result

**char str[6];**

**printf("Hi\n");**

**scanf("%5s", str);** //if you enter "HelloBello123xyz"

**printf("%s\n" , str);** //only the first 5 characters

//be read and a concluding '\0’

//will be put at the end

**Reading in Strings – scanf() (cont.)**

• scanf() reads in a string of characters, only up to the first nonwhitespace character

– it stops reading when it encounters a space, tab, or newline character

• C supports a format specification known as the edit set conversion code **%[..]**

– it can be used to read a line containing a variety of characters, including white spaces

char str[20];

printf("Enter a string:\n");

scanf("%[^\n]", str);

printf("%s\n",str);

• Always use the width field to limit the maximum number of characters to read with "%s" and "%[...]" in all production quality code!

– No exceptions!

**Reading in Strings – gets()**

• **gets()** is used to scan a line of text from a standard input device

• The gets() function will be terminated by a newline character

• The newline character won't be included as part of the string

• The string may include white space characters

‘\0’ is always appended to the end of the string of stored characters

char str[15];

printf("Enter your name: \n");

gets(str);

printf("%s\n", str);

• gets() has no provision for limiting the number of characters to read

– This can lead to overflow problems!

**Reading Strings Character by Character**

• Read in character by character is useful when

– you don't know how long the string might be,

– or if you want to consider other stopping conditions besides spaces and newlines

* periods, or when two successive slashes, //, are encountered.

• The scanf() format specifier for reading individual characters is %c

**– Here you must use the ‘&’ symbol!!!**

• If a width greater than 1 is given (%2c), then multiple characters are read, and stored in successive positions in a char array

**sscanf() and sprintf() functions**

• scanf() and printf() functions are used to read from and write to the standard input/output

• sscanf() and sprintf() are used for the same goal but instead of the standard input/output they use strings

• One of their main advantage is when you need to prepare a string for later use

– Examples in course exercise

**The ctype.h header**

• ctype.h declares a set of functions to classify and transform individual chars

– #include is required to use any of these functions

– http://www.cplusplus.com/reference/cctype/ documents the library

• Some of the more commonly used functions:

– isupper() – checks if a character is an uppercase letter

• A value different from zero is returned if the character is an uppercase alphabetic letter, zero otherwise

– islower() – checks if a character is a lowercase letter

• A value different from zero is returned if the character is a lowercase alphabetic letter, zero otherwise

– toupper() – converts a character to its uppercase equivalent if the character is an lowercase letter and has an uppercase equivalent

• If no such conversion is possible, the returned value is unchanged

– tolower() – converts a character to its lowercase equivalent if the character is an uppercase letter and has a lowercase equivalent

• If no such conversion is possible, the returned value is unchanged

**The string.h header**

• string.h defines several functions to manipulate null-byte terminated arrays of chars

– #include is required to use any of these functions

– http://www.cplusplus.com/reference/cstring/ documents the library

• Some of the more commonly used functions:

– strcpy() – returns the length of the string, not counting the ‘\0’

– strcat() – concatenates (appends) source to the end of destination

– strlen() – returns length of the string, not counting the ‘\0’

– strcmp() – compares strings str1 and str2, up until the first encountered null-term

• Returns zero if the two strings are equal

• Returns a positive value (1?) if the first encountered difference has a larger value in str1 than str2

• Returns a negative value (-1?) if the first encountered difference has a smaller value in str1 than str2

**The stdlib.h header**

• **stdlib.h** defines several functions, including searching, sorting and converting

– **#include<stdlib.h>** is required to use any of these functions

– http://www.cplusplus.com/reference/cstdlib/ documents the library

• Some of the more commonly used functions:

– atoi(), atof(), atol(), atoll() – parses a string of numeric characters into a number of type **int, double, long int, or long long int**, respectively

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:

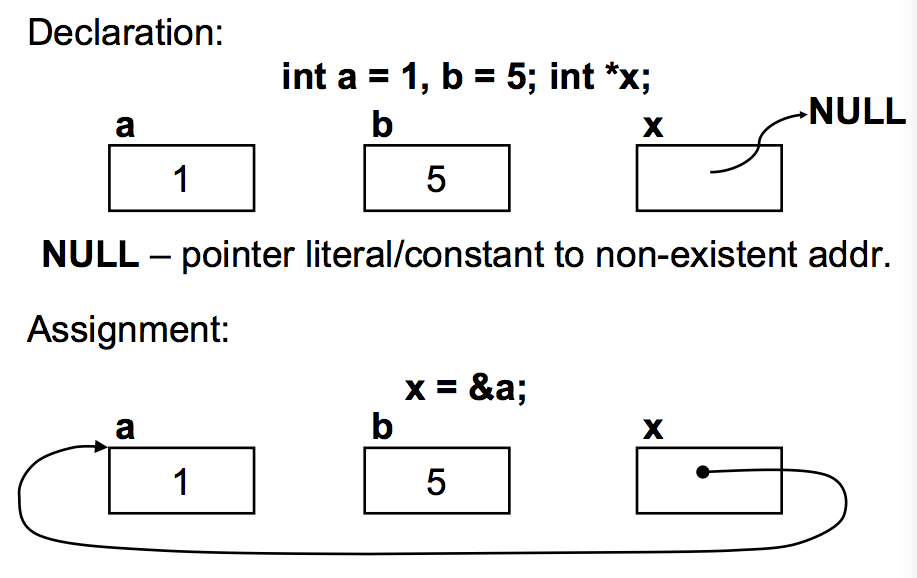
**int \*x;**

* 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;
2. there exists an intimate link between arrays and pointers in C.
3. To handle variable parameters passed to functions.
4. 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)

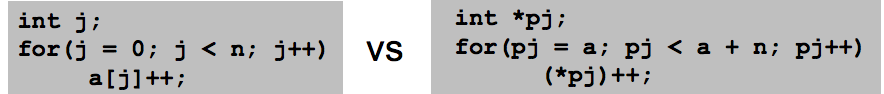
• Note that ip=ip+1 (or ip++) moves the pointer 4 bytes, instead of 1 to point to the next array element; amount added depends on size of array element

– 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

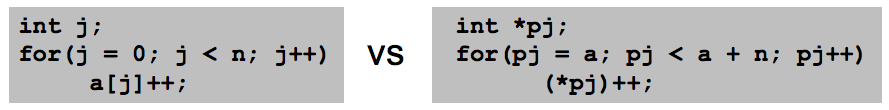
– a is a pointer to the beginning of an array of n elements; so a+n is the size of the array

– 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:

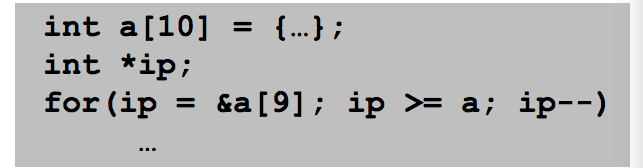
(\*pj)++; // increments what pj points to

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

• What do each of these do?

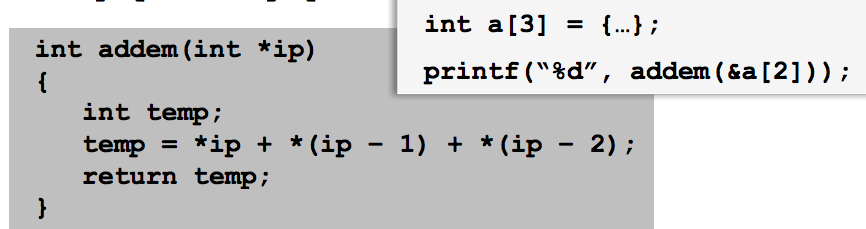
\*pj++; // unary ++ acts on pj, before \*indirection

++\*pj;

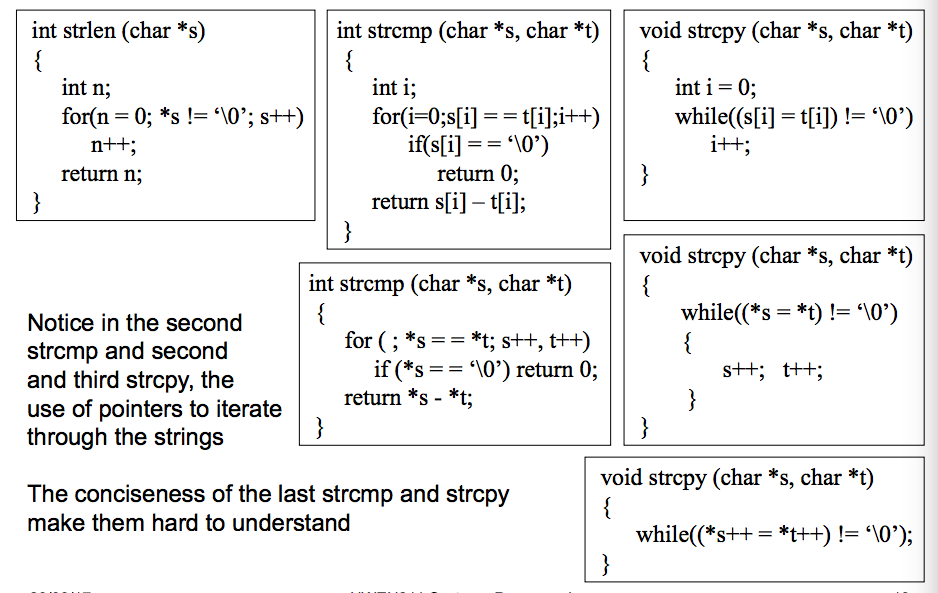


* Subtraction on pointers
* Example:

Pass to a function the address of the 3rd element of an array &a[2] and use pointer subtraction to get to a[0] and a[1].

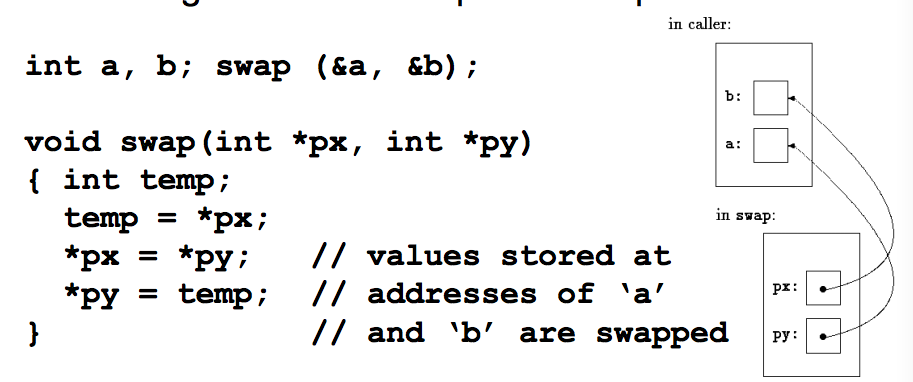


**Strings ❤ Pointers**



**Reference / variable parameters**

1. To make changes to a variable that exist after a function ends, we pass the address of (a pointer to) the variable to the function (a reference parameter)
2. Then we use indirection operator inside the function to change the value the parameter points to:



**Returning pointers from functions**

A function can also return a pointer value:

float \*findMax(float A[], int N) {

int I;

float \*theMax = &(A[0]);

for (I = 1; I < N; I++)

if (A[I] > \*theMax) theMax = &(A[I]);

return theMax;

}

void main() {

float A[5] = {0.0, 3.0, 1.5, 2.0, 4.1};

float \*maxA;

maxA = findMax(A,5);

\*maxA = \*maxA + 1.0;

printf("%.1f %.1f\n",\*maxA,A[4]);

}

**Returning pointers from functions**

Caution!!!

• In functions, do not do return p, where p is a pointer to a local variable.

Recall:

• local variables are deallocated when the function ends

– so whatever p is pointing to will no longer be available

– but if you return the pointer, then you still are pointing at that memory location even though you no longer know what is there

Question:

• Why is it allowed in the previous example?

**Pointer to pointers**

A pointer can also be made to point to a pointer variable (but the pointer must be of a type that allows it to point to a pointer)

**Example:**

int V = 101;

int \*P = &V; /\* P points to int V \*/

int \*\*Q = &P; /\* Q points to int pointer P \*/

printf(“%d %d %d\n”, V, \*P, \*\*Q); /\* prints 101 3 times \*/

**Pointer Types**

Pointers are generally of the same size (enough bytes to represent all possible memory addresses), but it is inappropriate to assign an address of one type of variable to a different type of pointer

**Example:**

int V = 101;

float \*P = &V; /\* Generally results in a Warning \*/

Warning rather than error because C will allow you to do this (it is appropriate in certain situations)

**Casting Pointers**

When assigning a memory address of a variable of one type to a pointer that points to another type, it is best to use the cast operator to indicate the cast is intentional (this will remove the warning).

**Example:**

int V = 101;

float \*P = (float \*) &V; /\* Casts int address to float \* \*/

Removes warning, but is still unsafe to do this !!!

**General (void) pointer**

A **void \*** is considered to be a general pointer

No cast is needed to assign an address to a **void \*** or from a **void \*** to another pointer type

**Example**:

int V = 101;

void \*G = &V; /\* No warning \*/

float \*P = G; /\* No warning, still unsafe \*/

Certain library functions return **void \*** results

**A useful quote to remember**

*“Pointers have been lumped with the goto statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity.” – Kernighan and Ritchie, 1988*

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

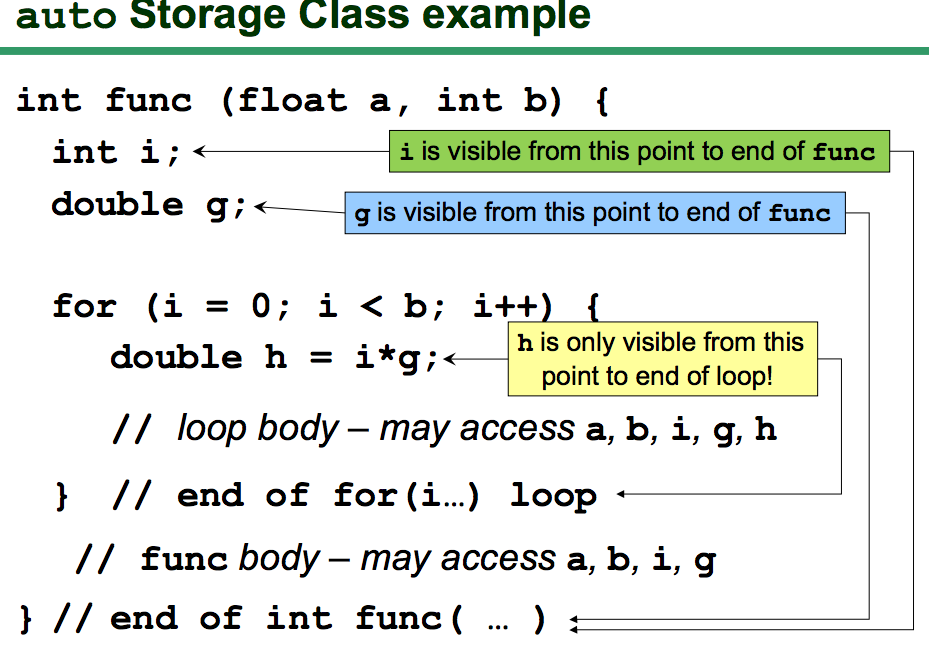
– Storage is released when function returns or block exits

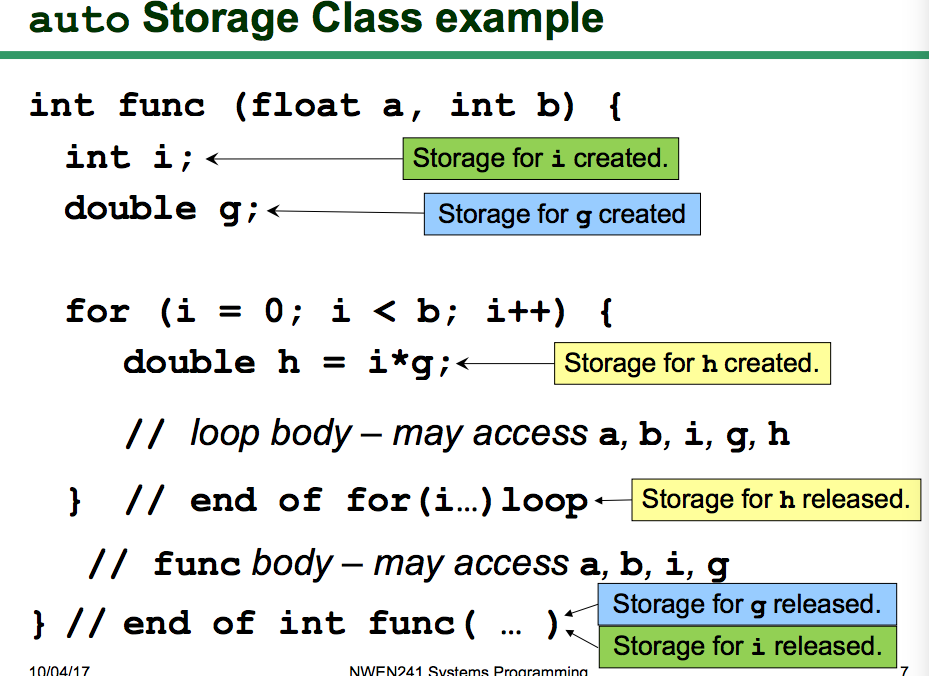
• Parameters and result are similar to automatic variables

– 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

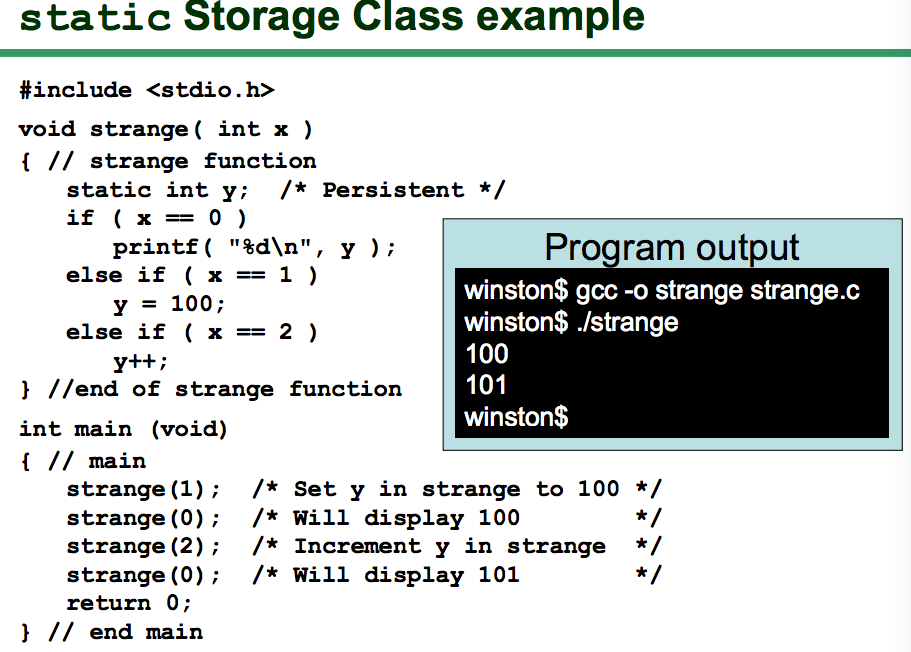
• If a static variable is simultaneously defined and initialized:

– 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) {**

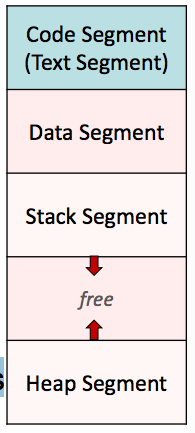
**printf("%f\n", x); /\* Access global x \*/**

**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

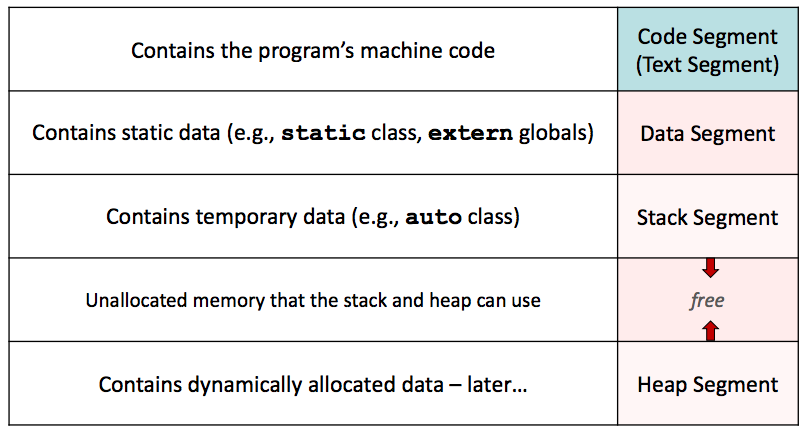
• Text / Code Segment

– 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;**

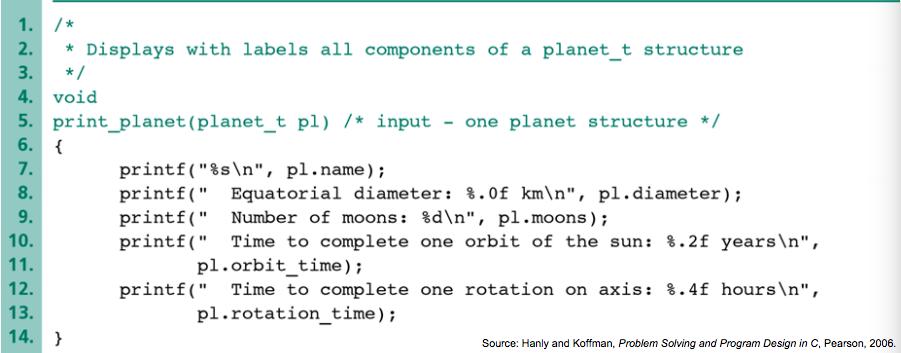
**int moons;**

**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)**

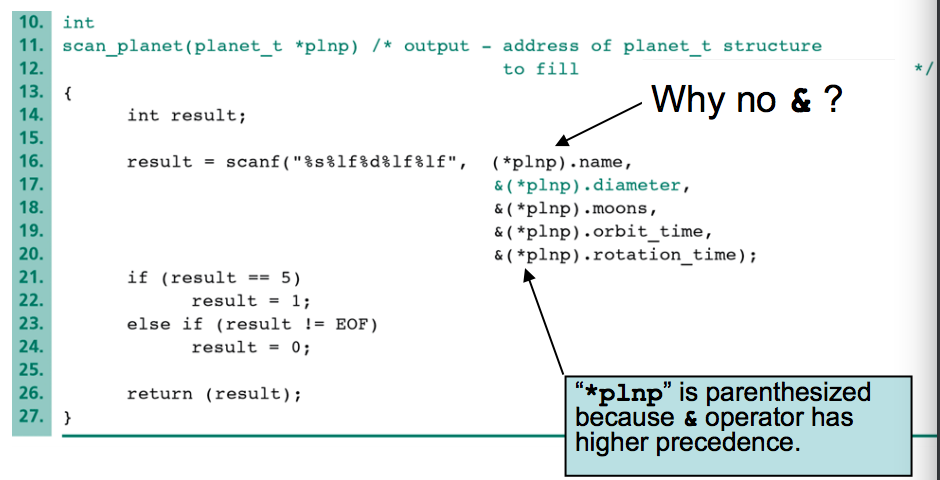
* If we define a variable as follows to store data to be read in:

**planet\_t current\_planet;**

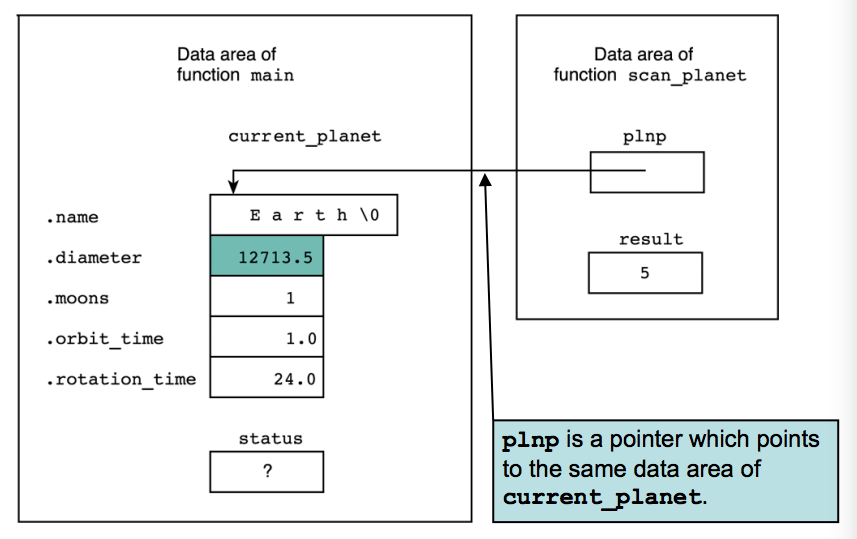
• For the following function, we call it by passing the parameter by reference:

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

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

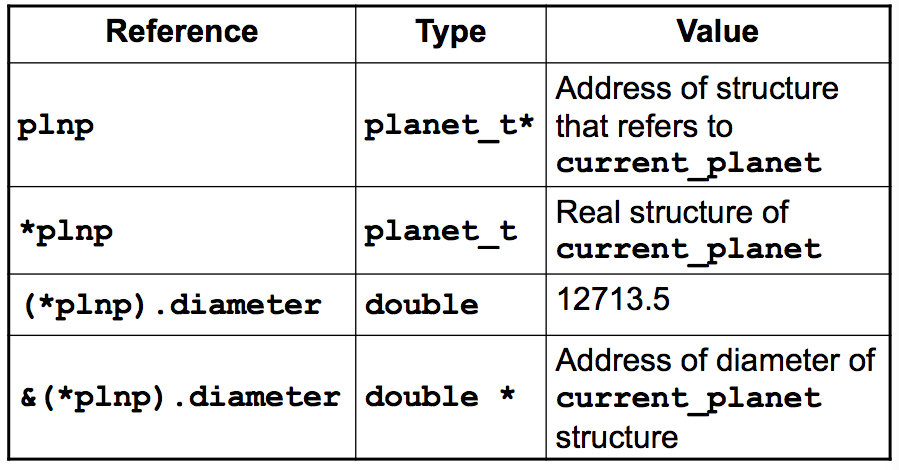


**Data Areas of function call**



**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.

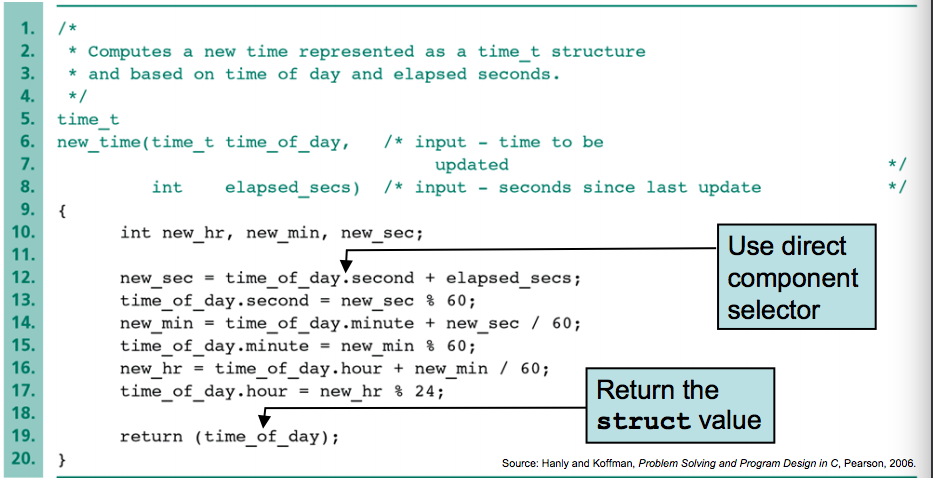
**&plnp->diameter**

is the same as

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

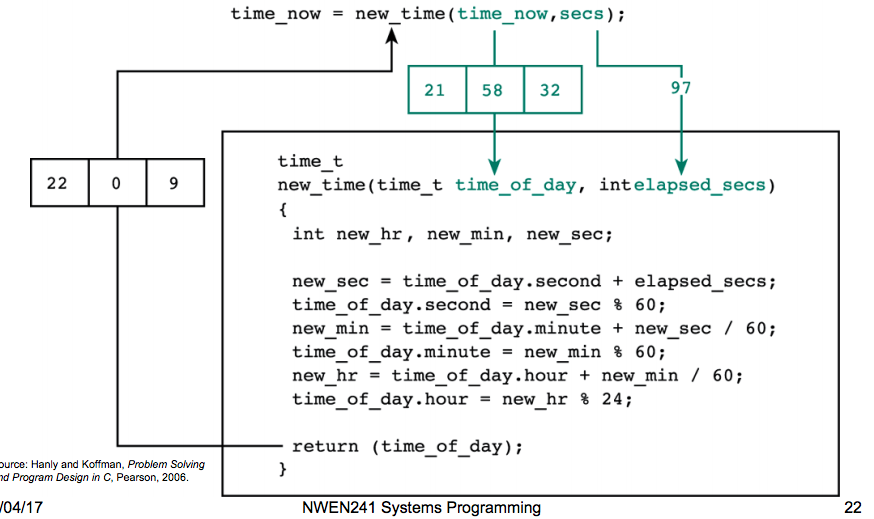
**Function returning a struct result type**

• 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;**

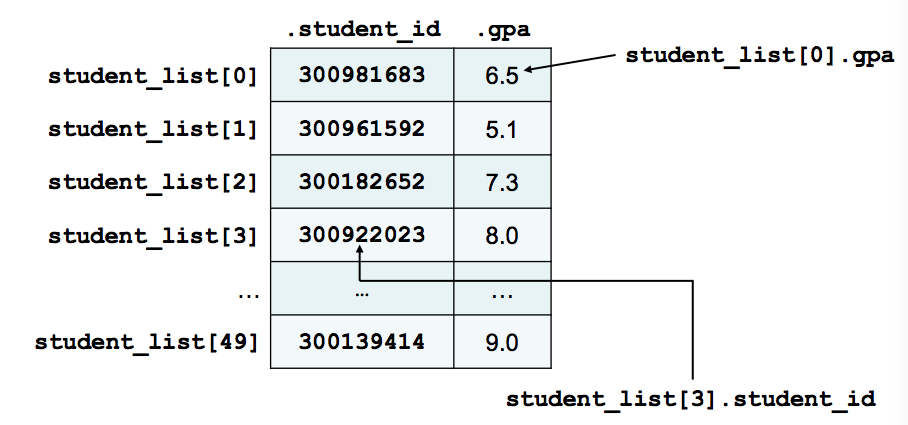
**student\_t student\_list[50];**

**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;

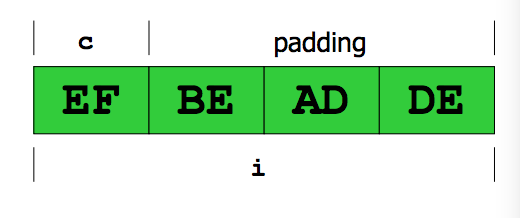
float f;

char c[4];

};

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

**union example**



union AnElt {

int i;

char c;

} elt1, elt2;

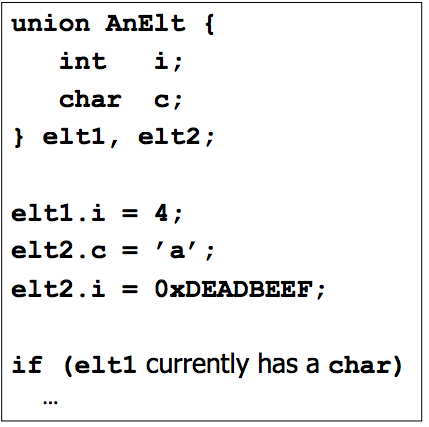
elt1.i = 4;

elt2.c = ‘ a‘

elt2.i = 0xDEADBEEF;

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

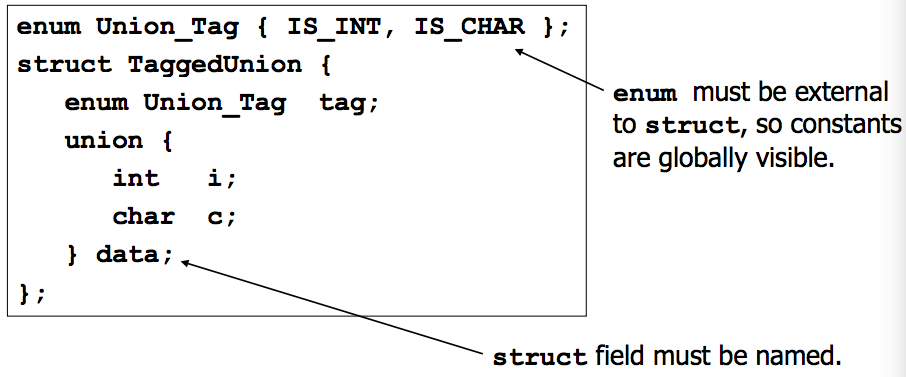
* How should your program keep track whether elt1, elt2 hold an int or a char?



• 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).**

Week8

Process vs Program

• Process – a program in execution

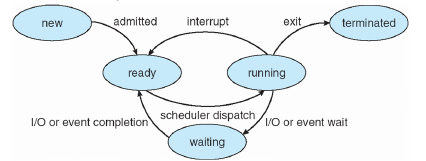
• A process includes (among other things):

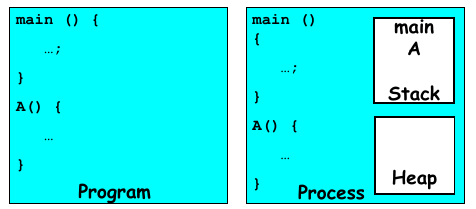
– program counter

– data section

– stack

• Process life-cycle





• Program is static, with the potential for execution

• Process is a program in execution and have a state

• One program can be executed several times and

thus has several processes

**Process Control Block (PCB)**

•Information associated with each process

–Process state

–Program counter

–CPU registers

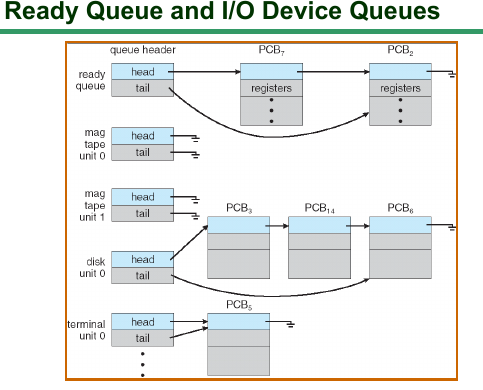
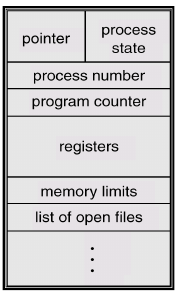
–CPU scheduling information

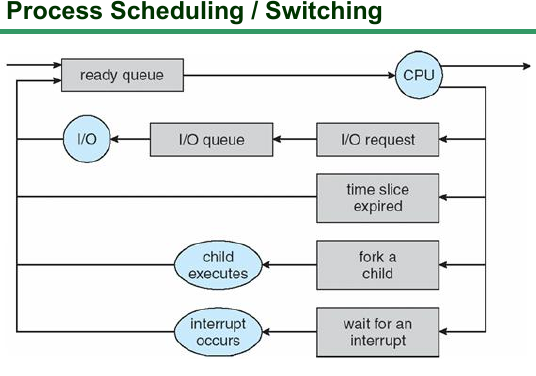
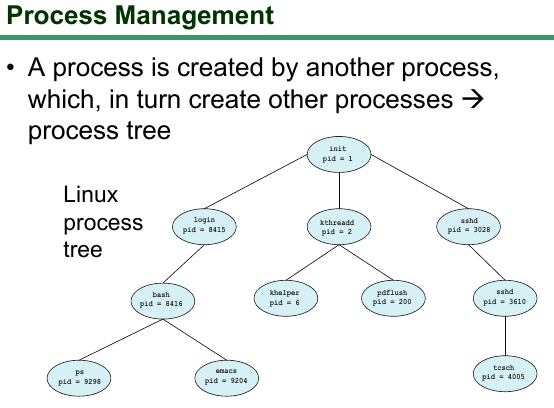
–Memory-management information

–Accounting information

–I/O status information

* A process is named using its process ID (PID) or process #
* Data is stored in a process control block





**Process Management**

•Parent and child process

– In Linux, using “ps –f”, the PPID field is the parent

– The first process is *init* having process ID 1

\*After creating a child, the parent may either wait for

it to finish or continue concurrently

•Process Management in C using System Calls

–fork()

–exec()

–wait()

–exit()

System Call

• A direct request to the operating system to do

something on behalf of the program

• Typically programs are executed in **user** mode

• System call allows a switch from **user** mode to

**kernel** mode

Code User Mode

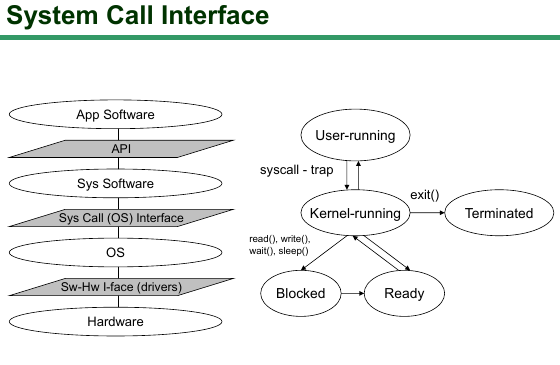
Code

System Call ---------→ Kernel Mode

Code

•The **kernel** is the core of the operating system for

managing **processes, files, networking**, etc..



**Process Creation with fork() System Call**

• A process calling **fork()** spawns a child process.

• After a successful **fork()** call, two copies of the original code will be running.

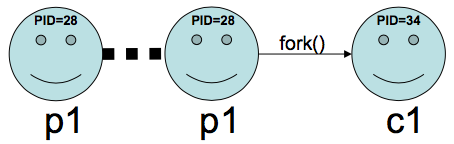
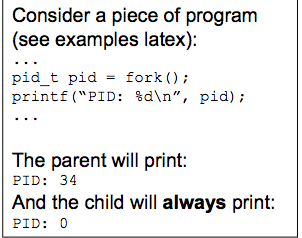
– Parent process – *return value* of **fork() –>** *child PID*.

– New child process – *return value* of **fork() ->** 0.

• **fork()** is called once, but returns twice!

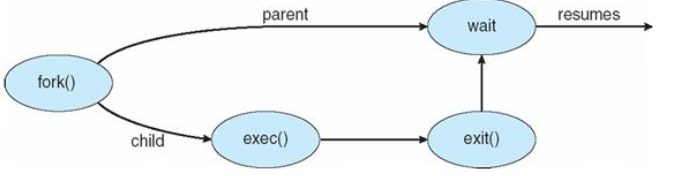
• After **fork()** both the parent and the child are executing the same program.

• On error, **fork()** returns -1.



**exec()System call (1)**

* The **exec()** call replaces a current process’image with a new one (i.e. loads a new program within current process).
* Upon success, exec() **never** returns to the caller. If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.
* Arguments passed via **exec()** appear in the **argv[]** of the **main()** function.



**exec() System call (2)**

There is no system call specifically by the name

**exec()**. By **exec()** we usually refer to a family of

calls:

– **int execl(char \*path, char \*arg, ...);**

– **int execv(char \*path, char \*argv[]);**

– **int execle(char \*path, char \*arg, ..., char \*envp[]);**

**– int execve(char \*path, char \*argv[],char \*envp[]);**

**– int execlp(char \*file, char \*arg, ...);**

**– int execvp(char \*file, char \*argv[]);**

• The various options*l*,*v*,*e*,and*p*mean:

– ***l*** : an argument list,

– ***v*** : an argument vector,

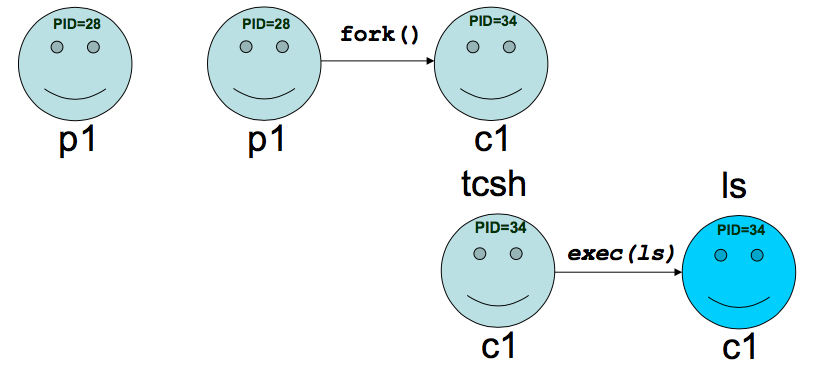
– ***e*** : an environment vector, and

– ***p*** : a search path.

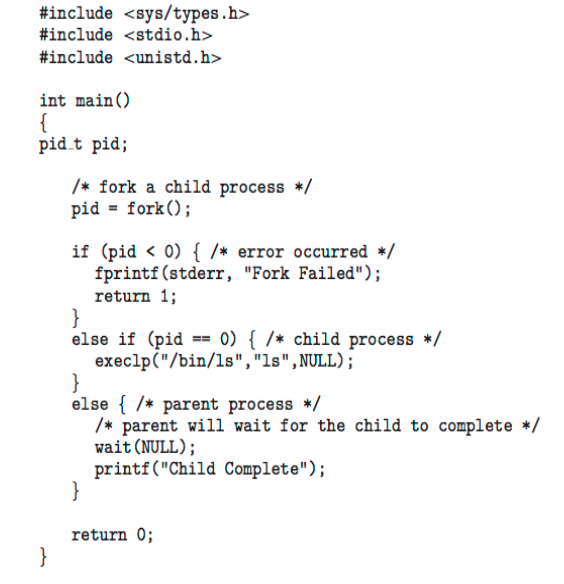
**fork() and exec() together**

• Often after doing **fork()** we want to load a new program into the child

• Most common e.g. a shell.



**Example of forking separate processes**



**wait() Call System**

Forces the parent to suspend execution,i.e.wait for its children or a specific child to die (*terminate* is

more appropriate terminology, but a bit less common).

**pid\_t wait(int \*status);**

**waitpid()** waits for the child with specific PID.

**pid\_t waitpid(pid\_t pid, int \*status, int options);**

The status, if not NULL, stores exit information of the child, which can be analyzed by the parent.

Thereturnvalueis:

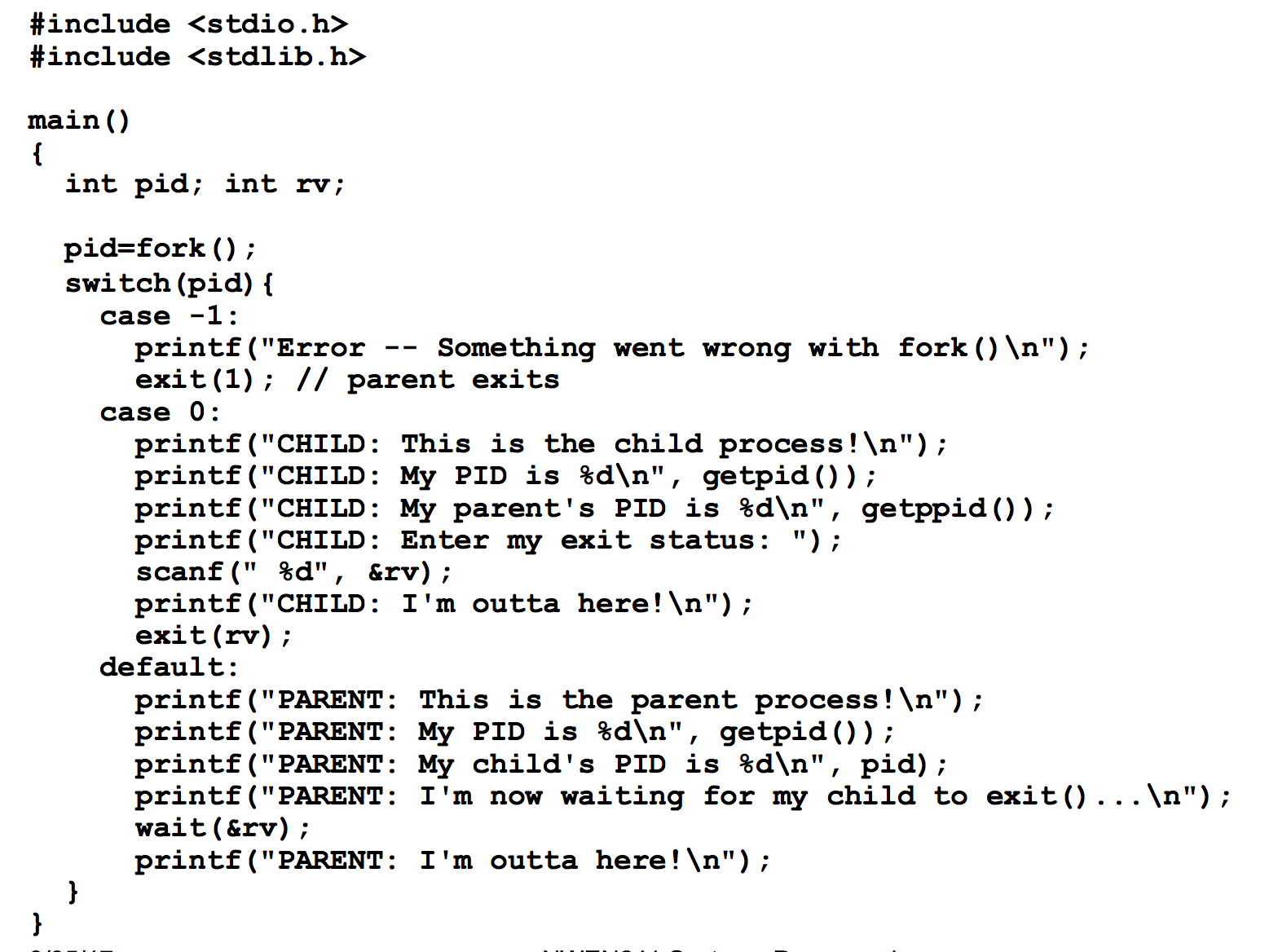
– PID of the exited process, if no error

– (-1) if an error has happened

**exit() System Call**

* Gracefully terminates process execution,meaning it does clean up and release of resources, and puts the process into the ***zombie*** state---> *terminated but still waiting for parent process to read its exit status*.
* By calling **wait()**,the parent cleans up all its zombie children.
* **exit()** specifies a return value from the program, which a parent process might want to examine as well as status of the dead process.
* **\_exit()** call is another possibility of quick death without cleanup.

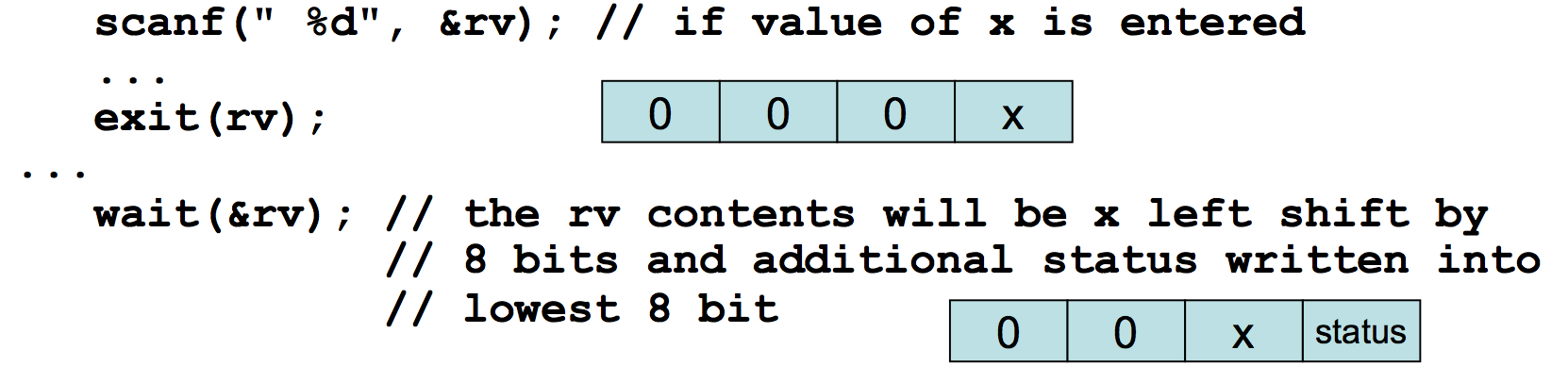
**Example of wait() and exit()**



**More about wait() and exit()**

• Should not interpret the status value of system call **wait(&status)** literally. If **&status** is not NULL, **wait()** stores status information in the **int** to which it points.

* Value returned by **exit(&status)** is moved to 2nd byte and 1st (lowest) byte is used to store the status information.
* In previous example:



**More about wait() and exit()**

This **status** integer can be inspected with macros (which take the integer itself as an argument, **not** a pointer to it):

–  **WIFEXITED(status)**

* –  **WEXITSTATUS(status)** This macro should be  employed only if **WIFEXITED** returned true.
* –  **WIFSIGNALED(status)**
* –  **WTERMSIG(status)** This macro should be employed only if **WIFSIGNALED** returned true.
* –  **WCOREDUMP(status)**
* –  **WIFSTOPPED(status)**
* –  **WSTOPSIG(status)**
* –  **WIFCONTINUED(status)**