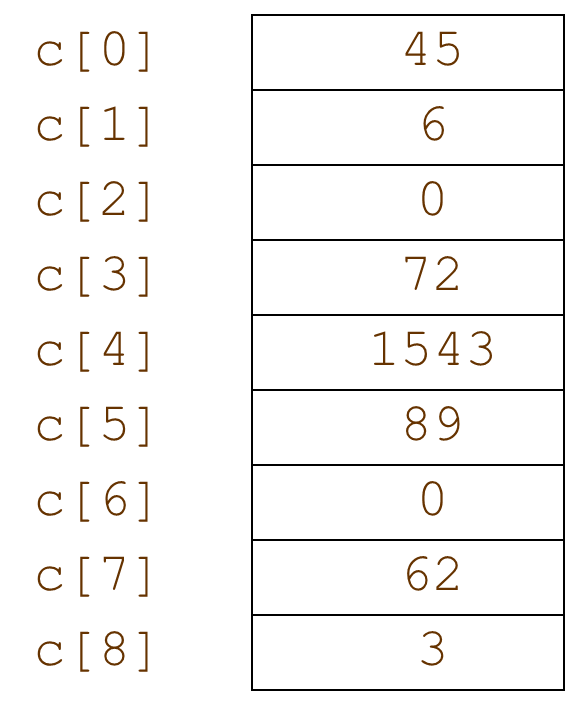
1. Simple Variables
   1. For data-rich programs, it would be limiting to express all data as variables.
   2. Each variable would have a different name.
      1. Impossible to use within a loop.
2. **Data Structures**: group data into sets of related data.
   1. Each variable does not have to be declared individually.
   2. Names created in some pre-planned manner to be easily used within loops.
   3. 2 ways
      1. **Arrays**: all data of the same type.
      2. **Structure**: data of different types.
3. **Arrays**: provide space for several **cells** (variables) of the same type.
   1. Numbered and ordered sequentially
   2. All variables in an array are located in **contiguous memory**: *sequential* memory.
      1. Advantages
         1. Use for pointer math
         2. Facilitates going from one cell to the next inside a loop
      2. Disadvantage: compiler must find a space in memory large enough for the whole array.
      3. Allocated statically: remain the same size throughout its scope.
   3. Name of each cell is the name of the array followed by an index.
   4. **Index number**: integer number that indicates the position in the set that each variable occupies.
      1. [0, infinity]
         1. Cannot be negative
      2. Syntax
         1. Array: c
         2. Cell 6 within array c: c [6]
         3. Same rules as for general variables
      3. Example



* 1. Arrangements
     1. Vector arrays: 1-dimensional
        1. Most common
     2. Matrix arrays: 2-dimensional
        1. Second most common
     3. *n*-dimensional
        1. Difficult to visualize with *n* > 3 dimensions
  2. Declaration
     1. <variable\_type> <arrayName> [<# of cells expressed as int>];
     2. Can be local or global
        1. Local preferred whenever possible
     3. Size of an array is typically done through a macro
        1. #define SIZE 10
  3. Variable Length Array (VLA)
     1. Example
        1. int x = 50;
        2. int list[x];
     2. Original C: not legal
     3. Newer implementations
        1. Work in theory
        2. Does not work well, and often makes program crash

1. Pointers and Arrays
   1. Name of an array: a pointer to its first element.
      1. Example
         1. Array a with 10 elements
         2. a == &a[0]
         3. a is a pointer to the element a[0]
      2. Can be used in pointer arithmetic
         1. a[3] can also be referenced as \*(a+3)
            1. 3 is the **offset** to the pointer
            2. Parentheses needed because precedence of \* is higher than that of +.

\*(a+3) 🡪 a[3]

Otherwise, \*a + 3 🡪 a[0] + 3

* + - 1. a+3 can be written as &a[3]
  1. Index: how many spaces away from the first element that cell is.
     1. a[1] is 1 space away from a[0]
  2. Size of each space: is the size of data type being stored in that array.
  3. Caveats
     1. Pointer arithmetic is meaningless outside of arrays.
        1. Nothing else is defined in contiguous memory.

1. Shortcuts when Declaring and Initializing Arrays
   1. C allows for shortcuts when declaring arrays
   2. Arrays are not automatically initialized
      1. Can be initialized during declaration or within a loop in the program
      2. Example
         1. int n[10] = {32,27,64,18,95,14,90,70,60};
         2. int n[] = {32,27,64,18,95,14,90,70,60};
            1. Automatically sets the length of the array to 10.
      3. If more elements than initialized: others = 0
      4. If less elements than initialized: compilation error
2. Carrying Array Elements
   1. Elements of an array can be used as variables.
   2. Examples
      1. int myarray[100];
      2. myarray[25]=98;
         1. Sets value of cell *26* to 98
3. Passing Arrays to Functions
   1. Arrays can be passed by reference since they are pointers.
   2. Passing array by a function
      1. The called function can modify the original array’s values.
      2. Pass array name without brackets, because **that is the pointer**.
         1. Address of the first element.
      3. Include the size as a passed value.
      4. Function header and prototype must indicate that an array is being received.
      5. Example
         1. void function1(int [], int); //takes an array of integer value
         2. void function2(int); //only receives one cell, not the entire array
         3. void main()
         4. {
         5. int a[5] = {0,1,2,3,4};
         6. function1(a, 5); //pass entire array name by reference
         7. function2(a[3]); //pass 1 cell by value
         8. }
4. Multi-Dimension Arrays
   1. Arrays can have an arbitrary number of dimensions indicated by bracket pairs.
   2. Example
      1. int a[5][10]; //1st is row; 2nd is column
      2. int b[10][12][20];
   3. Can be called in the same way as vector arrays.
   4. Initializing
      1. By row braces
         1. First brace equates to first row, 2nd to 2nd, etc.
         2. If missing value, C assumes 0
      2. Example
         1. int c[2][2] = {{1,2}{3,4}};
            1. Initializes

c[0][0]=1

c[0][1]=2

c[1][0]=3

c[1][1]=4

* + - 1. int c[2][2] = {{1}{3,4}};
         1. Initializes

c[0][0]=1

c[0][1]=0

c[1][0]=3

c[1][1]=4

* 1. Arrays and Strings
     1. **Strings**: arrays of characters.
        1. Each 8-bit cell of array contains one character.
        2. Pointers to the first element of the array.
     2. Mapped to ASCII table
     3. Characters are actually integer values.
     4. Always make character array 1 element longer than the number of elements between quotation marks.
        1. Last element is NULL

1. Arrays and Loops
   1. Closely-related
      1. Specially for loops, since arrays are generally counter-controlled
   2. Traversing through array (e.g., printing out contents of array)
      1. 1-dimensional array: single loop
         1. int myarray[200];
         2. int n;
         3. for(n = 0; n < 200 n++)
            1. printf(“The value of the array at cell %d is %d.\n”,n,myarray[n]);
      2. *n*-dimensional array: nested (at an arbitrary level) for loops
         1. int myarray[200][200];
         2. int n, m;
         3. for(n = 0; n < 200; n++)
            1. for (m = 0; m < 200; m++)

printf(“The value of the array at row %d and column %d is %d.\n”,n,m,myarray[n][m]);

* + 1. Start at 0 due to 0th element

1. Arrays of Pointers
   1. Arrays may contain any type of variable, including pointers.
      1. Usable to store a set of strings
   2. Example
      1. char \* suit[4] = {“hearts”, “diamonds”, “spades”, “clubs”};
      2. //char\* says that the elements of the array are pointers to char
   3. Arrays to character arrays
      1. Implicitly: 1-dimensional
      2. Explicitly: *n*-dimensional
2. Storing Data of Different Types
   1. Arrays: storing data of the same type.
      1. Easy…
         1. To access
         2. To assign value
         3. To put into loops
            1. Index is a variable
      2. Cumbersome if data are related but dissimilar in type.
   2. Structures: collection of related but dissimilar variables under one name.
      1. Advantages
         1. Greater flexibility than arrays
      2. Disadvantages
         1. More difficult to work with
      3. Uses
         1. Defining records to be stored in files
         2. Forming dynamic data types
            1. Linked lists
            2. Linked stacks
            3. Linked queues
      4. Definitions
         1. Declaration of *model*/*template*
            1. struct planet //keyword 🡪 structure tag
            2. { //Start of structure components
            3. char \*name; //Member 1; String (character array/pointer)
            4. int num\_moons; //Member 2
            5. double dist\_from\_sun; //Member 3
            6. float dist\_from\_earth; // Member 4
            7. };
         2. Cannot yet initialize variables, however
         3. Variable Type Definition
            1. Not limited to the type of variables we currently have.
            2. Defining new variable types

Example

typedef unsigned long int myvar;

Allows myvar to be used in place of unsigned long int

Usable for defining variable types that look like a structure template

Normally, the name of user-defined data has the first letter capitalized.

* + - * 1. **Instantiating**: creating an *instance* of the structure model.
        2. Creating structure variables
      1. Template vs. Variable
         1. Creating instance variables from a structure template: 3 ways

Instantiations in body

Instance names added after body definition of the structure template (i.e., between } and ;)

Structure tag is optional

Without it, it becomes anonymous

Disadvantage: since the structure has no name, you cannot instantiate variables later.

Example with structure tag

struct planet {

char name[10];

int num\_moons;

double dist\_from\_sun;

float dist\_from\_earth;

} earth, mars, solar[9], \*ptr; //solar[9] = array of 9 structures of data type *planet*

//\*ptr = pointer to data type

Example without structure tag

struct {

char name [10];

int num\_moons;

double dist\_from-sun;

float dist\_from\_earth;

} earth, mars, solar[9], \*ptr;

struct keyword used along with tag

Structure tag is obligatory

Example

struct planet{

charname[10];

int num\_moons;

float dist\_from\_earth;

double dist\_from sun;

};

…

struct planet earth

User-defined data type with typedef; to be instantiated later

Best to make it global

Just below preprocessor directives in the source code

Example requiring structure tag

typedef unsigned long int myvar;

typedef struct planet Planet;

Planet earth, mars;

Planet \* ptr;

Planet solar[9];

//structure tag is *planet*

Example not requiring structure tag

typedef struct planet{

char name[10];

int num\_moons;

double dist\_from\_sun;

float dist\_from\_earth;

} Planet;

//*planet* structure tag not needed

Without typedef; with structure tag

struct planet {

char name[10];

int num\_moons;

double dist\_from\_sun;

float dist\_from\_earth;

} earth, mars, solar[9], \*ptr; //solar[9] = array of 9 structures of data type *planet*; \*ptr = pointer to data type

Without typedef; Without structure tag

Disadvantage: since the structure has no name, the only way to instantiate variables is in the one and only definition of the template.

With typedef; With structure tag

typedef struct planet Planet

{

…

};

…

Planet earth, mars;

Planet \* ptr;

Planet solar[9];

//Assumes the structure definition includes a tag (“planet”) as before

//The name of the user-defined data type is “Planet”

With typedef; With Structure Tag (directly in definition)

typedef struct planet

{

char name[10];

int num\_moons;

double dist\_from\_sun;

float dist\_from\_earth;

} Planet;

//planet tag unnecessary

//**a variable can be instantiated** by writing “earth” after “Planet”

With Structure Tag; Without type def; In a different statement

struct planet{

charname[10];

int num\_moons;

float dist\_from\_earth;

double dist\_from sun;

};

…

struct planet earth

* + 1. **Members**: components of the structures.
       1. Can be accessed individually as variables after being instantiated
       2. Instantiating
          1. *Structure member operator* (*dot* *operator*): works on statically-allocated declared variables.

Example

mars.num\_moons = 1;

Directly accesses contents of num\_moons member in the mars structure

Sets to 1

* + - * 1. *Structure pointer operator* (*arrow operator*): works on pointers.

Example

mars\_ptr->num\_moons

Directly points to the contents of the num\_moons members

Same thing as…

(\*mars\_ptr).num\_moons

* + - 1. Initializing
         1. Just like arrays; use values inside curly brackets

If less values than members, only the first few are initialized.

Must be constant or evaluable expressions.

* + - * 1. Example

struct planet earth = {earth,1,1,.0e+6,0}

* + - * 1. If less values than members, only the first few are initialized.

Others = 0.

Must keep track of data types.

* + - 1. Structures and Functions
         1. Structures can be passed to functions…

By value

As individual structure members

As an entire structure variable

Function must know it is receiving a data type that is actually a structure.

By reference

As pointer to a structure

* + - 1. Template vs. Variable
         1. Creating instance variables from a structure template: 3 ways

Instantiations in body

Instance names added after body definition of the structure template (i.e., between } and ;)

Structure tag is optional

Without it, it becomes anonymous

Disadvantage: since the structure has no name, you cannot instantiate variables later.

Example with structure tag

struct planet {

char name[10];

int num\_moons;

double dist\_from\_sun;

float dist\_from\_earth;

} earth, mars, solar[9], \*ptr; //solar[9] = array of 9 structures of data type *planet*

//\*ptr = pointer to data type

Example without structure tag

struct {

char name [10];

int num\_moons;

double dist\_from-sun;

float dist\_from\_earth;

} earth, mars, solar[9], \*ptr;

struct keyword used along with tag

Structure tag is obligatory

Example

struct planet{

charname[10];

int num\_moons;

float dist\_from\_earth;

double dist\_from sun;

};

…

struct planet earth

User-defined data type with typedef; to be instantiated later

Best to make it global

Just below preprocessor directives in the source code

Example requiring structure tag

typedef unsigned long int myvar;

typedef struct planet Planet;

Planet earth, mars;

Planet \* ptr;

Planet solar[9];

//structure tag is *planet*

Example not requiring structure tag

typedef struct planet{

char name[10];

int num\_moons;

double dist\_from\_sun;

float dist\_from\_earth;

} Planet;

//*planet* structure tag not needed

* + - 1. Flexibility of Structures
         1. Structure member can be an array.

Strings

* + - * 1. Structure members can be other structures.

Nested structures

Example: scheduling system

typedef struct{

char model[12];

char delivery[10];

int place\_in\_line;

}Torque\_converter;

//Assembly for fluid injector

typedef struct{

char model[12]

char warehouse[10];

char date\_of\_manufacture;

}Fluid\_injector;

typedef struct{

Torque\_converter x;

Fluid\_injector y;

char delivery[12];

}Auto\_trans;

* + - * 1. Structure members can be *self-referencing structures*

Members that contain structures same as itself

Pointers that point to similar structures as itself

* + - * 1. There can be arrays of structures.

Example: Roster of 250 students

struct student{

char first\_name[15];

char last\_name[15];

int test\_grades[3];

int hw\_grades[8];

int final\_grade;

char ltr\_grade;

};

typedef struct student Roster;

Roster cop\_3223;

//each cell in array cop\_3223 has a student structure template

1. **Unions**: perform same role as structures, except members share memory space.
   1. Saves space when some members are never used at the same time.
      1. Not recommended except when memory management is needed.
      2. Example when two members are mutually-exclusive.
   2. Space for a member must be large enough to accommodate for the largest of data types to be stored in that member.
   3. Definition and declaration is similar to structure
      1. union
2. **Enumeration constants**: allow a set of integer constants to be represented by symbolic identifiers.
   1. Value can be set automatically.
      1. Start at 0 unless otherwise noted.
      2. Incremented by 1.
   2. Declaration
      1. enum
   3. Example
      1. #include <stdio.h>
      2. enum months {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC}
      3. main()
      4. {
      5. enum months month;
      6. char \* monthName[] = {“”, “January”, …};
      7. for (month =JAN;month<=DEC;month++)
      8. printf(“… .monthName[month];