UNIVERSITY of LOUISVILLE
dare to be great

C Chapter 9: Data Structures

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Data Structures: Motivation

- Arrays require that all elements be of the same data type.
- Many times it is necessary to group information of different data types. An example is a list of materials for a product. The list typically includes a name for each item, a part number, dimensions, weight, and cost.
- C and C++ support data structures that can store combinations of character, integer floating point and enumerated type data.
- They are called - *structs*.

Structures

- **struct** is used to declare a new data-type. Basically this means grouping variables together.
- Variables/Components may be of different types
- A single *struct* can store the data for one object. An array of *structs* can store the data for several objects.
- Components of a `struct` are called the members of the `struct`
- `struct` is a reserved word

Struct Syntax

- The general syntax of a `struct` is:

```
struct structName
{
    dataType1 identifier1;
    dataType2 identifier2;
       .
       .
       .
    dataTypen identifiern;
};
```

Declaring Structures

Does Not Reserve Space

```
struct my_example
{
    int label;
    char letter;
    char name[20];
} ;
/* The name "my_example" is called a
    structure tag    */
```

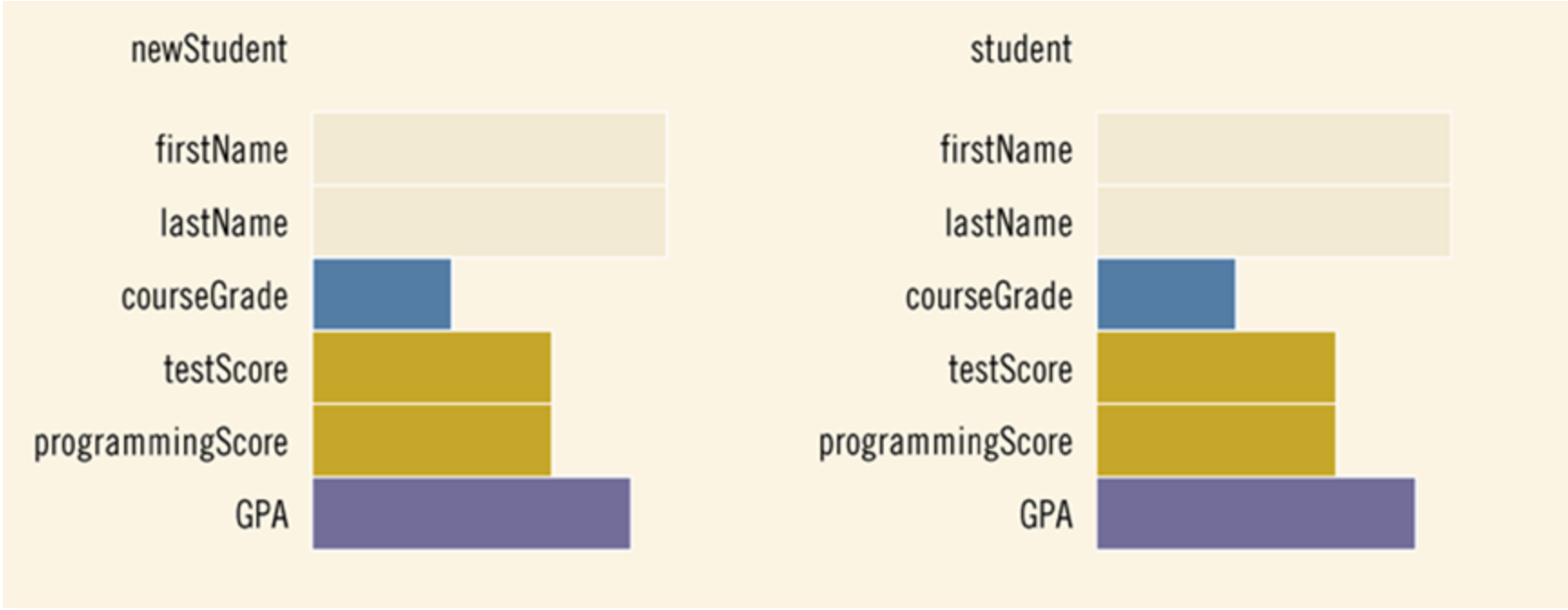
Reserves Space

```
struct my_example
{
    int label;
    char letter;
    char name[20];
} mystruct ;
```

struct Example

```
struct studentType
{
      string firstName;
      string lastName;
      char courseGrade;
      int testScore;
      int programmingScore;
      double GPA;
};

      //variable declaration
studentType newStudent;
studentType student;
```

structs newStudent and student



Accessing `struct` Members

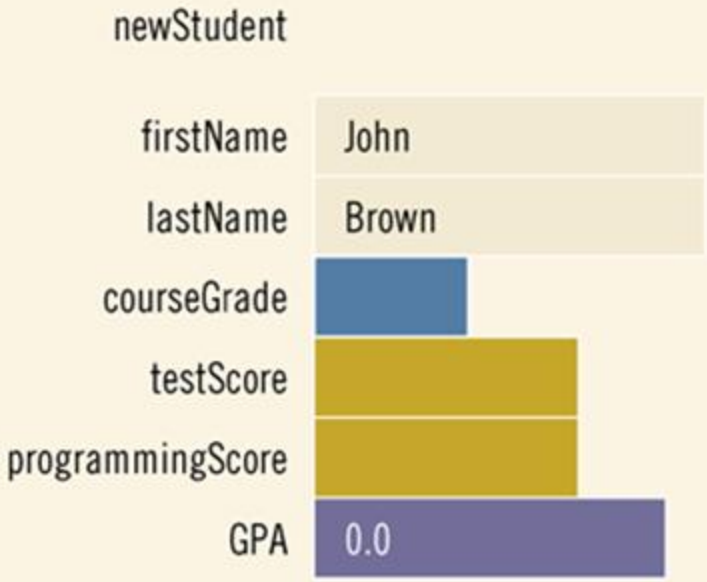
- The syntax for accessing a `struct` member is:

```
structVariableName.memberName
```

- The dot (`.`) is an operator, called the member access operator

Accessing `struct` Members

```
newStudent.GPA = 0.0;
newStudent.firstName = "John";
newStudent.lastName = "Brown";
```



Accessing struct Members

```
if (score >= 90)
    newStudent.courseGrade = 'A';
else if (score >= 80)
    newStudent.courseGrade = 'B';
else if (score >= 70)
    newStudent.courseGrade = 'C';
else if (score >= 60)
    newStudent.courseGrade = 'D';
else
    newStudent.courseGrade = 'F';
```

Assignment

- Value of one `struct` variable can be assigned to another `struct` variable of the same type using an assignment statement
- The statement:

```
student = newStudent;
```

copies the contents of `newStudent` into `student`

Assignment

The assignment statement:

```
student = newStudent;
```

is equivalent to the following statements:

```
student.firstName = newStudent.firstName;
student.lastName = newStudent.lastName;
student.courseGrade = newStudent.courseGrade;
student.testScore = newStudent.testScore;
student.programmingScore = newStudent.programmingScore;
student.GPA = newStudent.GPA;
```

Comparison (Relational Operators)

- Compare `struct` variables member-wise
- To compare the values of `student` and `newStudent`:

```
if (student.firstName == newStudent.firstName &&
    student.lastName == newStudent.lastName)
.
.
.
```

Input/Output

- No aggregate input/output operations on a struct variable
- Data in a `struct` variable must be read one member at a time
- The contents of a `struct` variable must be written one member at a time

Input/Output: Example

```
#include <stdio.h>

struct x {
      int a;
      int b;
      int c;
   };

   main()   {
      struct x z;

      z.a = 10;
      z.a++;

      printf(" first member is %d \n", z.a);

      system("pause");
      return 0;
}
```

User Defined Data Types (typedef)

- The C language provides a facility called *typedef* for creating synonyms for previously defined data type names.
- For example, the declaration:

```
typedef  int  Length;
```

makes the name *Length* a synonym (or alias) for the data type *int.*

- The data "type" name *Length* can now be used in declarations in exactly the same way that the data type *int* can be used:

```
Length a, b, len ;
Length numbers[10] ;
```

Typedef & Struct

- Often, *typedef* is used in combination with *struct* to declare a synonym (or an alias) for a structure:

```
typedef struct                  /* Define a structure */
{
   int label ;
   char letter;
   char name[20] ;
} Some_name ;                   /* The "alias" is Some_name */

Some_name  mystruct ;    /* Create a struct variable */
```

Accessing Struct Members with Pointers

- Individual members of a *struct* variable may be accessed using the structure member operator (the dot, “.”):

  mystruct.letter;

- Or , if a pointer to the *struct* has been declared and initialized

  Some_name *myptr = &mystruct ;

  by using the structure pointer operator (the “->“):

  myptr -> letter;

  which could also be written as:

  (*myptr).letter ;

Sample Program With Structs

```
#include <stdio.h>
struct personal {          //Create a struct but don't reserve space.
       long id;
    float gpa;
};

struct identity { //Create a second struct that includes the first one.
       char name[30];
    struct personal person;
};

int main ( ) {

   struct identity js = {"Joe Smith"}, *ptr = &js;
   js.person.id = 123456789;
   js.person.gpa = 3.4;
   printf ("%s %ld %f\n", js.name, js.person.id, js.person.gpa) ;
   printf ("%s %ld %f\n", ptr->name, ptr->person.id,ptr->person.gpa) ;
   system("pause");
}
```

Arrays VS. Structs

Aggregate Operation	Array	struct
Arithmetic	No	No
Assignment	No	Yes
Input/output	No (except strings)	No
Comparison	No	No
Parameter passing	By reference only	By value or by reference
Function returning a value	No	Yes

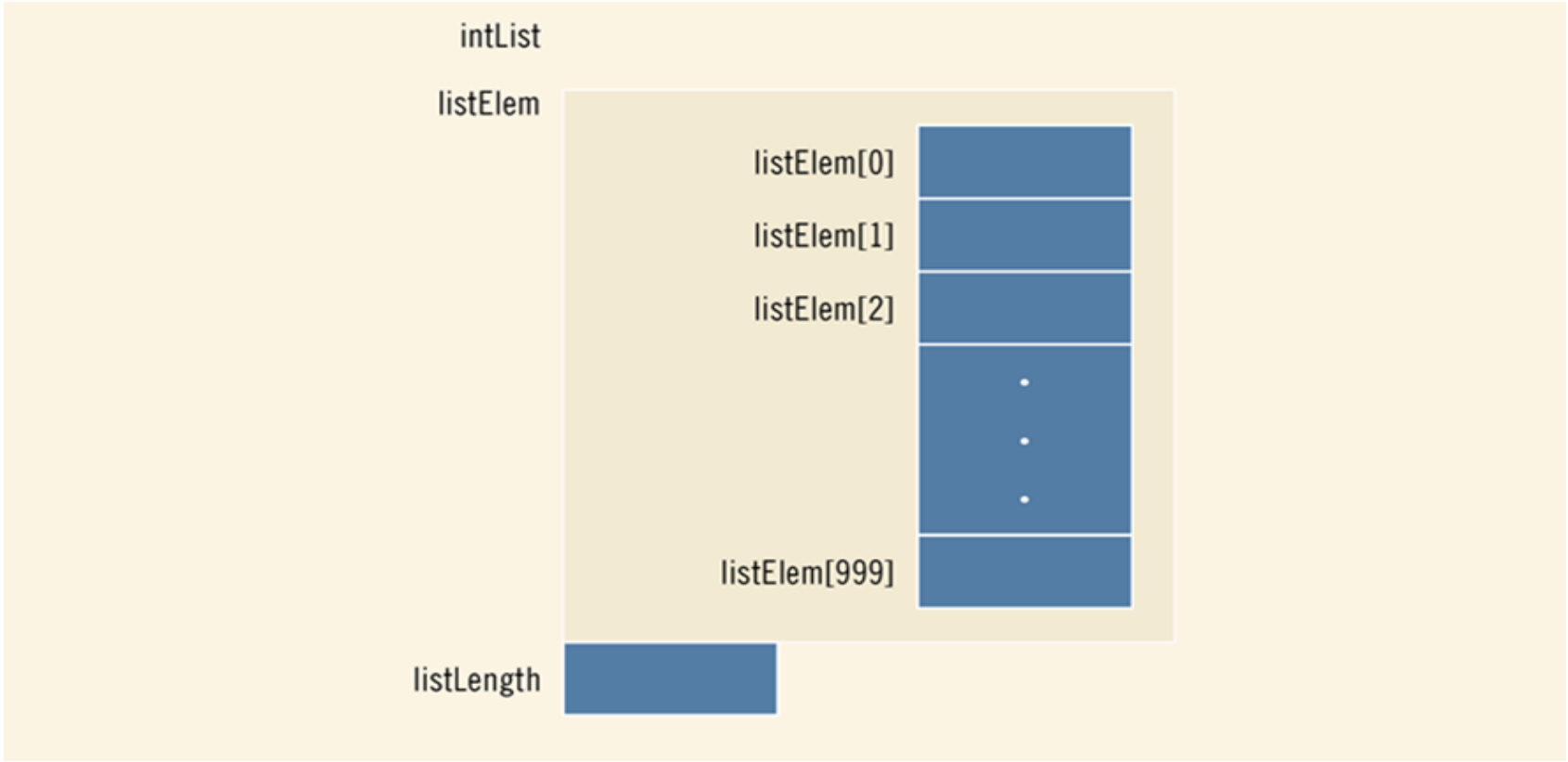
Arrays in structs: lists

- Two key items are associated with a list:
  - Values (elements)
  - Length of the list
- Define a struct containing both items:

```
const int ARRAY_SIZE = 1000;

struct listType
{
    int listElem[ARRAY_SIZE];     //array containing the list
    int listLength;               //length of the list
};
```

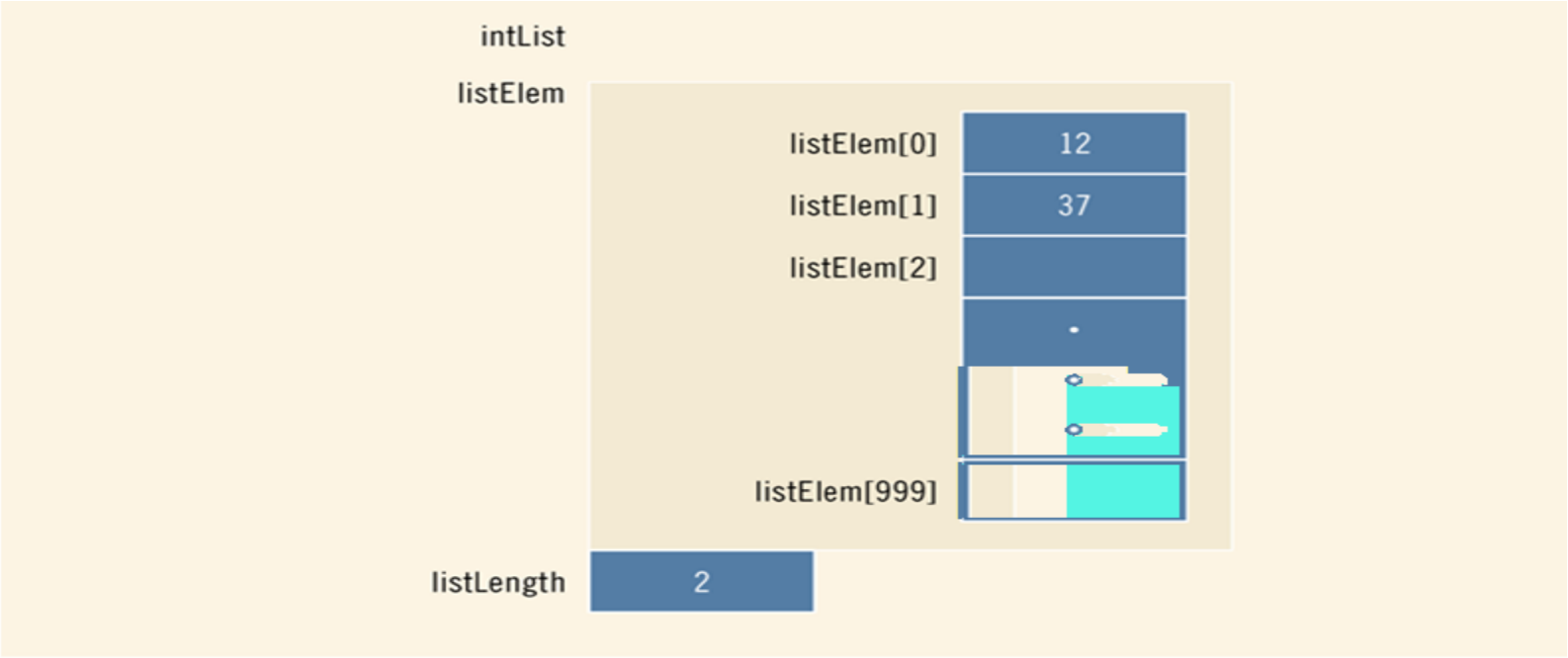
List struct

```
listType intList;
```



List struct

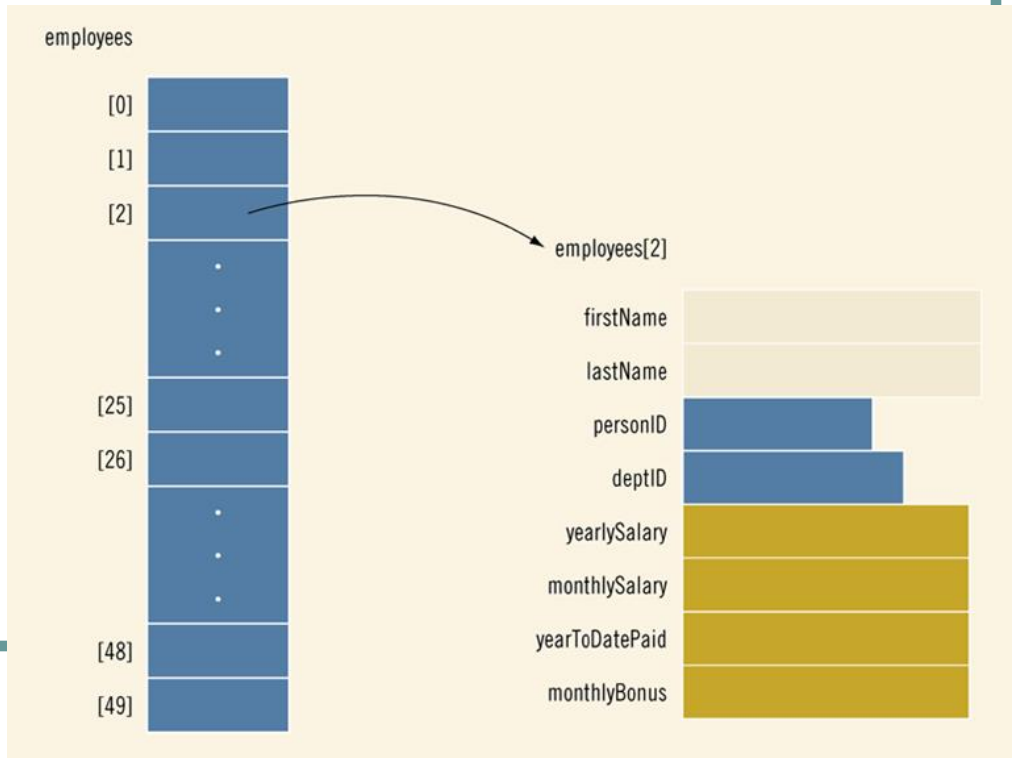
```
intList.listLength = 0;          //Line 1
intList.listElem[0] = 12;        //Line 2
intList.listLength++;            //Line 3
intList.listElem[1] = 37;        //Line 4
intList.listLength++;            //Line 5
```



Array of structs

```
struct employeeType
{
    string firstName;
    string lastName;
    int    personID;
    string deptID;
    double yearlySalary;

     double monthlySalary
     double yearToDatePaid;
     double monthlyBonus;
};
```

```
employeeType employees[50];
```



Structs within a struct

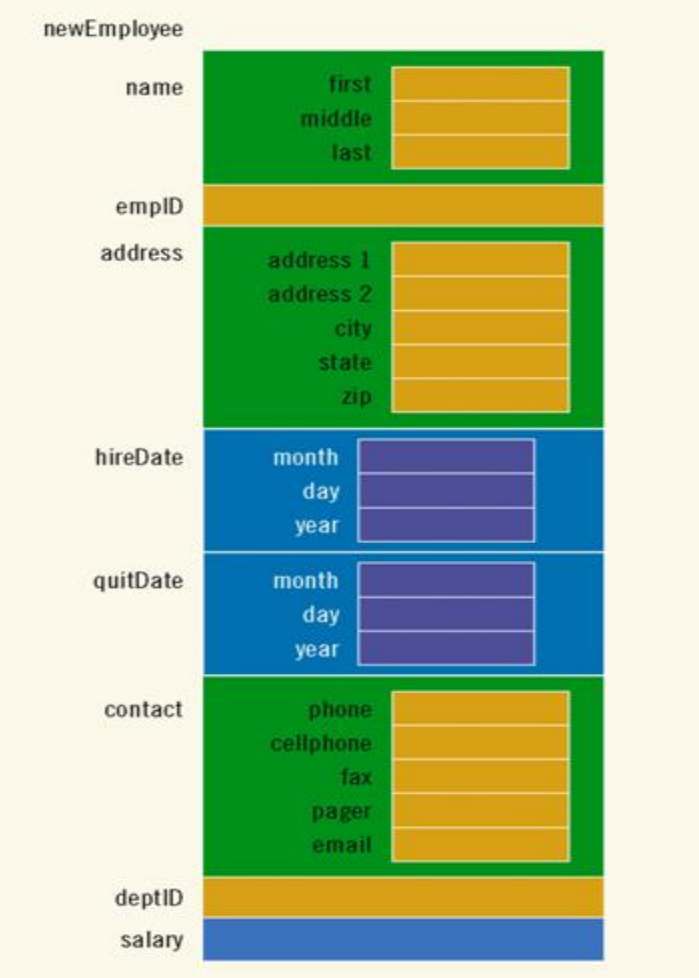
```
struct  employeeType
{
     string firstname;
     string middlename;
     string lastname;
     string empID;
     string address1;
     string address2;
     string city;
     string state;
     string zip;
     int hiremonth;
     int hireday;
     int hireyear;
     int quitmonth;
     int quitday;
     int quityear;
     string phone;
     string cellphone;
     string fax;
     string pager;
     string email;
     string deptID;
     double salary;
};
```

```
struct nameType
{
     string first;
     string middle;
     string last;
};

struct addressType
{
     string address1;
     string address2;
     string city;
     string state;
     string zip;
};

struct dateType
{
     int month;
     int day;
     int year;
};

struct contactType
{
     string phone;
     string cellphone;
     string fax;
     string pager;
     string email;
};
```

```
struct employeeType
{
     nameType name;
     string empID;
     addressType address;
     dateType hireDate;
     dateType quitDate;
     contactType contact;
     string deptID;
     double salary;
};
```

Structs within a struct

```
employeeType newEmployee;
```



Structures as Function Arguments

- Structures are scalars, so they can be returned and passed as arguments – just like int, char

```
struct BIG changestruct(struct BIG s);
```

  - Call by value: temporary copy of structure is created
  - Caution: passing large structures is inefficient

    – involves a lot of copying

- avoid by passing a pointer to the structure instead:

```
void changestruct(struct BIG *s);
```

- What if the struct argument is read-only?
  - Safe approach: use const

```
void changestruct(struct BIG const *s);
```

Structures as Function Arguments: Examples

- You may pass data fields of a structure to a function by value as we did with other data types. This will pass a copy of the value of the variable.

```
double calcNet(float payRate, float hours);
…
calcNet(salesPerson.payRate, salesPerson.hours);
…
```

- You may also pass a variable which is a structure. This will pass individual copies of the values of each of the data fields of the structure.

```
double calcNet(struct Employee);  /* function prototype */
…
netPay = calcNet(emp);    /* pass copies of the values in emp */
```

- You may also pass structures by reference. This passes the address of the structure.

```
double calcNet(struct Employee *);    /* function prototype */
…
netPay = calcNet(&emp);       /* pass an address */
```

In this case we refer to the data fields of the structure inside the function by using

```
ptToStructure->dataField  (most popular) which is equivalent to
    (*ptToStructure).dataField
```

Unions

- Like structures, but every member occupies the same region of memory!
  - Structures: members are “working” together
  - Unions: members are “working” one at a time

```
union VALUE {
  float f;
  int i;
  char *s;
};
/* either a float or an int or a string */
```

Unions

- Up to programmer to determine how to interpret a union (i.e. which member to access)
- Often used in conjunction with a “type” variable that indicates how to interpret the union value

```
enum TYPE { INT, FLOAT, STRING };
struct VARIABLE {
  enum TYPE type;
  union VALUE value;
};
```

Access `type` to determine how to interpret `value`

Unions

- Storage
  - size of union is the size of its largest member
  - avoid unions with widely varying member sizes;

    for the larger data types, consider using pointers instead

- Initialization
  - Union may only be initialized to a value appropriate for the type of its first member

Structs with Union

- The program on the next slide creates a union and makes it a member of struct *personal* which is, in turn, a member of struct *identity*.
- The union uses the same memory location for either rank or a character string (*deg*) depending on the answer to the prompt for student status in main( )

Structs with Union (cont.)

```
#include <stdio.h>

union status {
  int rank ;
  char deg[4] ;
};
struct personal {
  long id ;
  float gpa ;
  union status level ;
};
struct identity {
  char name[30] ;
  struct personal student ;
};
```

```
int main( ) {
   struct identity jb = {"Joe Brown"}, *ptr = &jb;
   char u_g;
   jb.student.id = 123456789 ;
   jb.student.gpa = 3.4 ;
   printf ("Enter student status - u or g\n");
   scanf ("%c", &u_g);
   if (u_g == 'u'){
       printf ("Enter rank -- 1, 2, 3, 4 or 5\n");
       scanf ("%d", &jb.student.level.rank);
       printf ("%s is level %d\n" , jb.name , jb.student.level.rank);
     } else {
      printf ("Enter degree sought -- ms or phd\n");
      scanf ("%s", &jb.student.level.deg);
      printf ("%s is a %s candidate\n", jb.name , jb.student.level.deg );
   }
   printf ("%s %ld %f\n" , jb.name , jb.student.id, jb.student.gpa );
   printf ("%s%ld %f\n" , ptr->name , ptr->student.id, ptr->student.gpa );
   system("pause");
}
```

Type Casting

- Forces one variable of a certain type to be another type
- Might be useful to force integer division
- Example:

```
int x = 12;
int y = 5;
float result = 0;
result = (float) x / (float) y;
```

- Output: 2.4, without type casting 2

Type Casting: characters, numbers

- You can also type cast numbers to characters
  - And characters to numbers

```
#include <stdio.h>

main() {
      int number = 86;
      char letter = 'M';
      printf("\n86 type-casted to char is: %c\n", (char)number);
      printf("\n'M' type-casted to an int is: %d\n", (int)letter);

     system("pause");
                return 0;
}
```

Output: 86 type-casted to char is: V
'M' type-casted to an int is: 77

The End!