BITS Pilani
Dubai Campus

Principles of Programming Languages

CS F301

innovate achieve lead

BITS Pilani
Dubai Campus

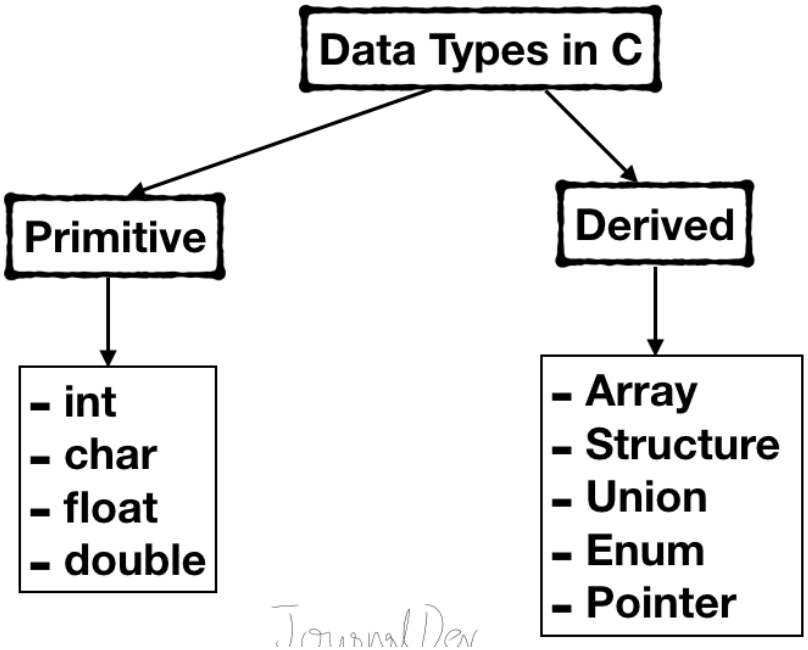
Types: Data Representation

Contents

- Introduction
- Primitive data types
- Array and Array address calculation
- Other non-primitive data types
    - Structure
    - Union
    - Enum
    - Pointers
- Dangling pointers and memory leak
- Types and error checking

Types: Data Representation

- Data type Definition
  - Collection of data objects
  - A set of predefined operations
  - Descriptor : collection of attributes for a variable.
  - Object :instance of a user-defined (abstract data) type
- Classification of Data types



Primitive Data Types in C

Data Type	Range	Bytes	Format
signed char	-128 to + 127	1	%c
unsigned char	0 to 255	1	%c
short signed int	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-32768 to +32767	2	%d
unsigned int	0 to 65535	2	%u
long signed int	-2147483648 to +2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-3.4e38 to +3.4e38	4	%f
double	-1.7e308 to +1.7e308	8	%lf
long double	-1.7e4932 to +1.7e4932	10	%Lf

Source: ://www.thecrazyprogrammer.com/2015/01/data-types-in-c.html

Type: Boolean

- Range of values: true | false
- Could be implemented as bits, but often as bytes
- Boolean types are often used to represent flags in programs.
- A Boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte.

Subrange Type

- A subrange type defines a subset of the values of a particular type.
- By using subrange types, you can easily detect errors occurring due to unreasonable values of a variable, which shouldn't take values outside a given boundary.
- code inserted (by the compiler) to restrict assignments to subrange variables

```
type
    digit=0..9;
    letter='A'..'Z';
var
    num:digit;
    alpha:letter
```

Arrays

- An array is a collection of data items, all of the same type, accessed using a common name.
- A one-dimensional array is like a list; A two dimensional array is like a table; The C language places no limits on the number of dimensions in an array, though specific implementations may.
- Some texts refer to one-dimensional arrays as ***vectors***, two-dimensional arrays as ***matrices***, and use the general term ***arrays*** when the number of dimensions is unspecified .
- Examples:

```
int i, j, intArray[ 10 ], number;
float floatArray[ 1000 ];
int tableArray[ 3 ][ 5 ];      /* 3 rows by 5 columns */
```

Calculate Memory Addresses in 2D Arrays

Row-major Implementation

- In this method, the first row elements are placed first, then the second row elements and so on.
- The formula to calculate the address of [I, J] th block is:

Address of [I, J]th element in **row-major** =

$$B + W[C(I - L_r) + (J - L_c)]$$

- B is the base address (address of the first block in the array).
- W is the width in bytes (size in bytes for each block in the array).
- L_r is the index of the first row.
- L_c is the index of the first column.
- C is the total number of columns.

Calculate Memory Addresses in 2D Arrays

Column-major Implementation

- In this method, the first column elements are placed first, then the second column elements and so on.
- The formula to calculate the address of [I, J] th block is:

Address of $[I, J]^{th}$ element in column-major =

$$B + W[R(J - L_c) + (I - L_r)]$$

- B is the base address (address of the first block in the array).
- W is the width in bytes (size in bytes for each block in the array).
- L_r is the index of the first row.
- L_c is the index of the first column.
- R is the total number of rows.

Example: Array Address Calculation

In C language representation, Calculate the address of A[5][5] in a 2D Array A[10][25] when it follows i) Row-major ordering

ii) Column-major ordering. Consider the following assumptions: W – 4Bytes; B – 1000.

i) Row-major:

$A[I][J] = B + W[C(I - L_r) + (J - L_c)]$

A[5][5] = 1000 + 4 * [25 (5-0) + (5-0)]

= 1000 + 4 * [125 + 5]

= **1520**

i) Col.-major:

$A[I][J] = B + W[R(J - L_c) + (I - L_r)]$

A[5][5] = 1000 + 4 * [10 (5-0) + (5-0)]

= 1000 + 4 * [50 + 5]

= **1220**

B=1010

W = 4

Indices A[5][5]

A[10][7] in a 2d array of size A[20][17]

Row major

$A[I][J] = B + W[C(I - L_r) + (J - L_c)]$

A[10][7] = 1010 + 4 [17(10 – 5) + (7 – 5)] =

Column major

A[I][J] = B + W[R(J-Lc)+(I-Lr)] = 1010+4[20(7-5)+(10-5)]=

Implementation of Array Types

- C, C++ uses row major.
- Implementing arrays requires considerably more compile-time effort.
- The code to allow accessing of array elements must be generated at compile time.
- At run time, this code must be executed to produce element addresses
- Dynamic allocation of arrays: Unlike a fixed array, where the array size must be fixed at compile time, dynamically allocating an array allows us to choose an array length at runtime.
- Java has built-in dynamic arrays.

Structures

- Structures
  - **Structure** allows to combine data items of different kinds.
  - Structures are used to represent a record.
  - Syntax in C

```
struct [structure name] {
   member definition;
   member definition;
   ... member definition;
} [one or more structure variables];
```

  - Example

```
struct Books {
      char title[50];
      char author[50];
      char subject[100];
      int book_id; } book;
```

Structures

- Structures (contd)

```
#include <stdio.h>
#include <string.h>
struct Books {
       char title[50];
       char author[50];
       char subject[100];
       int book_id; };
int main( ) {
       struct Books Book1; /* Declare Book1 of type Book */
       struct Books Book2; /* Declare Book2 of type Book */  /*
book 1 specification */
       strcpy( Book1.title, "C Programming");
       strcpy( Book1.author, "Nuha Ali");
       strcpy( Book1.subject, "C Programming Tutorial");
       Book1.book_id = 6495407;
```

Structures

```
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;
/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);
/* print Book2 info */
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id); return 0;
}
```

```
Output:
Book 1 title : C Programming Book 1 author : Nuha Ali Book 1
subject : C Programming Tutorial Book 1 book_id : 6495407
Book 2 title : Telecom Billing Book 2 author : Zara Ali Book 2
subject : Telecom Billing Tutorial Book 2 book_id : 6495700
```

Union

- A union is a special data type available in C that allows to store different data types in the same memory location.
- Syntax

```
union [union name] {
        member definition;
        member definition;
        ... member definition;
} [one or more union variables];
```

- Example

```
union Data {
        int i;
        float f;
        char str[20]; } data;
```

Union

```c
#include <stdio.h>
#include <string.h>
union Data {
      int i;
      float f;
      char str[20]; };
int main( ) {
      union Data data;
      data.i = 10;
      data.f = 220.5;
      strcpy( data.str, "C Programming");
      printf( "data.i : %d\n", data.i);
      printf( "data.f : %f\n", data.f);
      printf( "data.str : %s\n", data.str);
      return 0;
}
```

Output:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Enum

- It is used to assign names to the integral constants which makes a program easy to read and maintain.
- Syntax

```
enum enum_name{const1, const2, ....... };
```

- Example

```
#include<stdio.h>
enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
    return 0;
}
```

Output:

2

Pointers

- A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.
- Syntax: (pointer variable declaration)
  - `type *var-name;`
  - Example declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

Pointers

```c
#include <stdio.h>
int main () {
int var = 20; /* actual variable declaration */
int *ip; /* pointer variable declaration */
ip = &var; /* store address of var in pointer variable*/
printf("Address of var variable: %x\n", &var );
/* address stored in pointer variable */
printf("Address stored in ip variable: %x\n", ip );
/* access the value using the pointer */
printf("Value of *ip variable: %d\n", *ip );
return 0; }
```

Output:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Dangling Pointer vs Memory Leak

If a **pointer** is pointing to memory that is not owned by your program (except the **null pointer**) or an invalid memory, the pointer is called a dangling pointer.

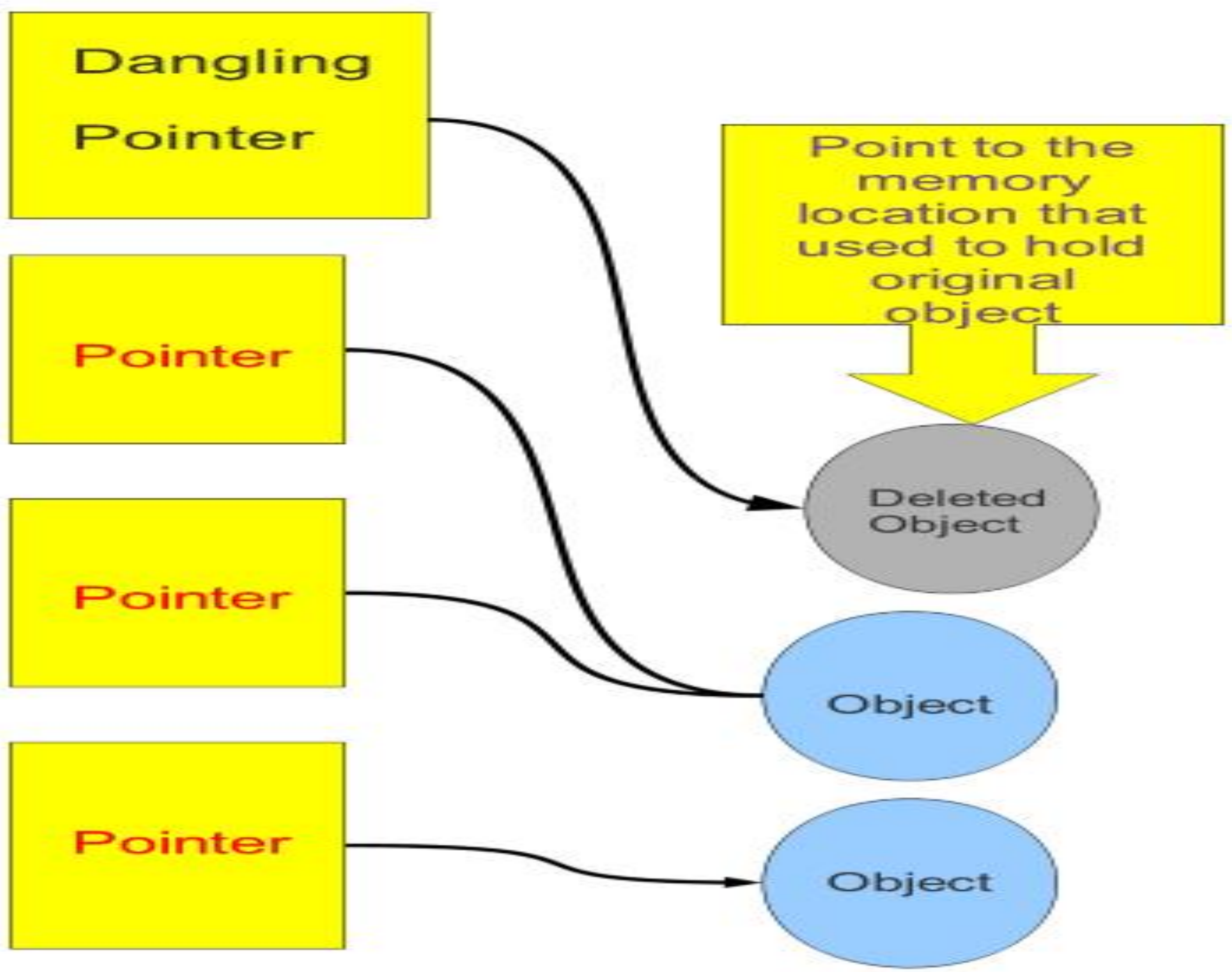
Generally, dangling pointers arise when the referencing object is deleted or deallocated, without changing the value of the pointers.

Null pointer is a pointer which is pointing to nothing. Null pointer points to empty location in memory. Value of null pointer is 0. We can make a pointer to point to null as below.

```
int *p =  NULL;   char *p = NULL;
```

A memory leak occurs when you forget to deallocate the allocated memory. In C language compiler does not deallocate the memory automatically it is freed by the programmer explicitly.

Dangling Pointer



Dangling Pointer: Example

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
int *p= NULL;

p= malloc(sizeof(int)* 5); //Allocate memory for 5 integer

free(p); //free the allocated memory

*p= 2; //p is dangling pointer

printf("%d",*p);

return 0;

}
```

Memory Leak: Example

In below program, programmer forgets to free the allocated memory, it can cause a memory leak.

```
int main()
{

   char * pBuffer = malloc(sizeof(char) * iLenBuffer);

   /* Do some work */

   return 0; /*Not freeing the allocated memory*/
}
```

Avoiding Dangling Pointer, Memory Leak in C

How to avoid creation of the dangling pointer in C?

We can easily prevent the creation of dangling pointer to assign a NULL to the freed pointer.

```
#include<stdio.h>
#include<stdlib.h>

int main()
{
char *pcData = NULL;

pcData = malloc(sizeof(char)* 10); //creating integer of size 10.

free(pcData); /* pcData can be becomes a dangling pointer */

*pcData = NULL; //pcData is no longer dangling pointer

return 0;
}
```

Avoiding Dangling Pointer, Memory Leak in C

How to avoid memory leaks in C?

Create a counter to monitor allocated memory

- It is a good technique to prevent the memory leaks.
- In this technique, we will create two global counters and initialize them with 0.
- In every successful allocation, we will increment the value of the counter1 (Allocate_Counter)
- and after the deallocating the memory we will increment the counter2 (Deallocate_Counter).
- At the end of the application, the value of both counters should be equal.This method helps you to track the status of allocated memory.

Every malloc or calloc should have a free function

Types & Error Checking

- Infer the type of variable and values
- Useful for type conversion, error checking.
- Avoid inadvertent errors in programs.
- How safe the program is? (eg bufferoverflow)

- Variable Bindings
  - Compiled languages : variables have a fixed type./values are dynamic
  - Interpreted languages: variable type can change./values are dynamic
  - Associate a property with a variable
  - Static binding/early binding
  - Dynamic binding/late bindings
  - Inheritance and Polymorphism

Types & Error Checking

- Type Systems
  - Set of rules for associating a type with expressions
  - Int I; i=i+2; int a=1; int b=4; float c; c = a+b;
  - Can be used for detecting invalid operations on incompatible types
    - Adding a pointer to integer is ok.
    - Adding 2 pointers is incorrect.

- Type Checking. void add(int a, int b); add(12,12.5)
  - Uses the property of a function.
  - Functions maps element of set A to set B
    - Eg. Arithmetic operations are functions
      - E + F : if E and F have type int then E + F has type int.
      - Overloading of built in operators.

Types & Error Checking

- Coercion : Implicit type Conversion
  - Original Fortran type system rejected expressions of the form E + F where E is of type int and F is of type real
  - Now PLs automatically do the conversion (implicit conversion)
    - 2 + 3.14 is considered as 2.0 +3.14

  e.g. double x, y;

  x = 3;   // implicitly coercion (coercion)

  y = (double) 5;   // explicitly coercion (casting)

- Polymorphism
  - Static Polymorphism
    - Functions/operator level (overloading)
      - C allows for buitlin type and operators
      - C++ allows for user-defined functions and data types.(templates)
    - Compile time.
    - No run time overheard

Types & Error Checking

- Polymorphism
  - Run Type Polymorphism
    - The method call is decided at run time.
      - Shape a = new Rectangle(a,b,c,d);
      - a.draw();
      - Run time overheads.
      - How does compiler generate code?

Types & Error Checking

- Type Name and Type Equivalence
  - When do say that 2 variables are of the same type.
  - Required to check if the assignment is valid.
  - Structural Equivalence.
    - SE1: A type name is structurally equivalent to itself
      - Eg: char is equivalent to char, float to float
    - SE2: 2 types are structurally equivalent if they are formed by applying the same constructor to structurally equivalent types
      - Eg. char ≡ char , So char[10] ≡ char[10]
    - SE3: After type declaration (typedef X Y), the type name X is SE to Y

Types & Error Checking

- Restricted Type Equivalence.
  - Pure name equivalence:
    - Type name is equivalent to itself
  - Transitive name equivalence :

    typedef XX int; typedef YY XX;typedef ZZ int

    Then XX, YY, ZZ are equivalent
  - Type expression equivalence :
    - Type name is equivalent only to itself.
    - Apply same constructor to equivalent expressions.

    - C: Uses SE for all type except structures.
    - Circular Types: Link List specifically circular link list.

Static & Dynamic Type Checking

- Type checking ensure operations in a program are applied properly
- So no errors (logical errors), eg if pointer addition is allowed.
- Type error: defined as ..
  - If a function/operation expects an arg of type T and is supplied an arg of S which is not SE to T.
  - A + “123” , not OK in C , but Ok in Java
- Type safe program: that which executes w/o type errors.
- Static Type checking : Compile Time.
- Dynamic Type checking: Run Time. Increases execution time.
- Strongly typed lang.
  - Strong : accept only safe expressions

Sources

- Chapter 4, Ravi Sethi, "Programming Languages: Concepts and Constructs" 2nd Edition by Addison Wesley, 2006.
- https://www.tutorialspoint.com/cprogramming
- https://aticleworld.com/dangling-pointer-and-memory-leak/

innovate achieve lead

BITS Pilani
Dubai Campus

Thank you.