**Unit 5**

**Structures**

A structure is a collection of variables of different of same or different data types.

**Syntax of declaration of structure:**

struct struct\_name

{

Data\_type1 member1;

Data\_type2 member2;

…………………………………..

 data\_typen member;

};

* struct is a keyword
* strut\_name is the name of the structure which is user defined
* datatype1,datatyp2,…..datatypen are the basic data types
* member1,member2,…..membern are the different data items

Example:

struct book

{

   int pages;

   char author [30];

   float price;

};

**Structure variables**

There are three ways of declaring structure variables, which are as follows –

|  |  |  |
| --- | --- | --- |
| **Type 1**  struct book  {     int pages;     char author[30];     float price;  }b; | **Type 2**  Struct  {     int pages;     char author[30];     float price;  }b; | **Type 3**  struct book  {     int pages;     char author[30];     float price;  };  struct book b; |

**Structure declaration using typedef**

#include<stdio.h>

typedef struct Point

{

int x;

int y;

} hello; // hello is an alias for struct Point

int main()

{

hello p1; // p1 is a variable of type hello(struct Point)

p1.x = 1;

p1.y = 3;

printf("%d \n", p1.x);

printf("%d \n", p1.y);

return 0;

}

**Structure declaration without using typedef**

#include<stdio.h>

struct Point

{

int x;

int y;

} a;

int main()

{

a.x = 1;

a.y = 3;

printf("%d \n", a.x);

printf("%d \n", a.y);

return 0;

}

**Initialization of structures**

Initializing structures in C can be done using various methods.

|  |
| --- |
| **Method 1: Initializing Each Member Individually** You can initialize each member of the structure individually by assigning values to them one by one:  #include <stdio.h>  struct Point  {  int x;  int y;  } ;  int main()  {  // Initializing each member individually  struct Point p1;  p1.x = 1;  p1.y = 3;  printf("p1: %d, %d\n", p1.x, p1.y);  return 0;  } |
| **Method 2: Initializing the Entire Structure at Once** You can initialize the entire structure at once by providing values in curly braces:  #include <stdio.h>  struct Point  {  int x;  int y;  } ;  int main()  {  // Initializing the entire structure at once  struct Point p1 = {4, 6};  printf("p1: %d, %d\n", p1.x, p1.y);  return 0;  } |
| **Method 3: Using Designated Initializers** Designated initializers allow you to initialize specific members of a structure by explicitly specifying the member names. This method allows you to specify which member is being initialized by name, making the code more readable.  #include <stdio.h>  struct Point {  int x;  int y;  };  int main() {  // Initializing member using Designated initializers  struct Point p1 = {.x = 7};  // Print values  printf("p1: %d, %d\n", p1.x, p1.y);  return 0;  } |

**Accessing members of a structure**

Accessing members of a structure is done using the dot (**.**) operator. The dot operator allows you to refer to a specific member of a structure.

#include <stdio.h>

#include <string.h>

struct Person

{

char name[50];

int age;

float height;

};

int main()

{

struct Person p1;

strcpy(p1.name, "John"); **// p1.name= "John"; is invalid**

p1.age = 25;

p1.height = 6.1;

printf("Name: %s\n", p1.name);

printf("Age: %d\n", p1.age);

printf("Height: %.2f\n", p1.height);

return 0;

}

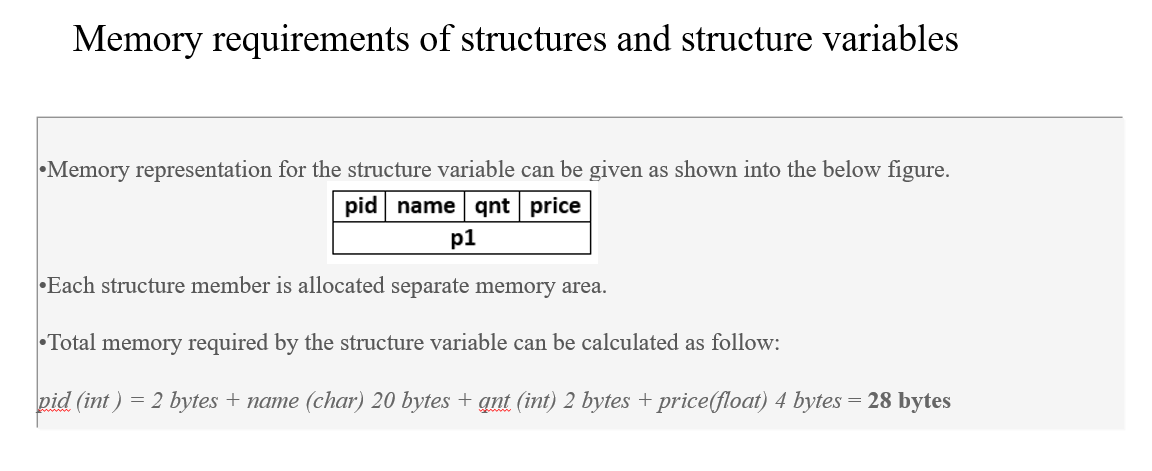
**Structure within structure**

You can have structures within structures, which is often referred to as nested structures. This allows you to create complex data structures with a hierarchical organization.

**Syntax**

|  |
| --- |
| struct outer\_struct  {  // outer structure members  int outer\_member1;  float outer\_member2;  **// nested structure definition**  struct inner\_struct  {  // inner structure members  int inner\_member1;  float inner\_member2;  } inner;  }outer;  Example  #include <stdio.h>  #include <string.h>  struct Employee  {  int id;  char name[20];  struct Date  {  int dd;  int mm;  int yyyy;  }doj;  }e1;  int main( )  {  //storing employee information  e1.id=101;  strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array  e1.doj.dd=10;  e1.doj.mm=11;  e1.doj.yyyy=2014;    //printing first employee information  printf( "employee id : %d\n", e1.id);  printf( "employee name : %s\n", e1.name);  printf( "employee date of joining (dd) : %d\n", e1.doj.dd);  printf( "employee month of joining (mm) : %d\n", e1.doj.mm);  printf( "employee date of joining (yyyy) : %d\n", e1.doj.yyyy);  //printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e1.doj.yyyy);  return 0;  } |
| struct inner\_struct  {  // inner structure members  int inner\_member1;  float inner\_member2;  } inner;  struct outer\_struct  {  // outer structure members  int outer\_member1;  float outer\_member2;  struct inner\_struct var\_name; **// nested structure variable**  }outer;  Example  #include<stdio.h>  struct employee  {  char name[50];  int id;  int age;  };  struct organisation  {  char name[50];  int id;  struct employee emp;  };  int main(){  struct organisation org1 = {"XYZ Ltd.", 123, {"John Doe", 101, 30}};  printf("Organisation name: %s\n", org1.name);  printf("Organisation ID: %d\n", org1.id);  printf("Employee name: %s\n", org1.emp.name);  printf("Employee ID: %d\n", org1.emp.id);  printf("Employee age: %d\n", org1.emp.age);  return 0;  } |

|  |  |
| --- | --- |
| **Copying one structure to another**  #include <stdio.h>  #include <string.h>  struct A  {  char name[50];  int age;  } ;  // Function to print the details of a person  void printPerson(struct A p)  {  printf("Name: %s\n", p.name);  printf("Age: %d\n", p.age);  }  int main()  {  // Initialize a person  struct A person1 = {"Alice", 30};  // Copy person1 to person2  struct A person2 = person1;  // Change person2's details  person2.age = 35;  strcpy(person2.name, "Bob");  // Print details of both persons  printf("Person 1:\n");  printPerson(person1);  printf("Person 2:\n");  printPerson(person2);  return 0;  }  **Output:**  Person 1:  Name: Alice  Age: 30  Person 2:  Name: Bob  Age: 35 | #include <stdio.h>  #include <string.h>  struct A  {  char name[50];  int age;  } ;  struct B  {  char name[50];  int age;  } ;  void printdata(struct A p)  {  printf("Name: %s\n", p.name);  printf("Age: %d\n", p.age);  }  void printdata1(struct B p)  {  printf("Name: %s\n", p.name);  printf("Age: %d\n", p.age);  }  int main()  {    struct A person1 = {"Alice",30};  struct B person2 = {"Bob",35};    printdata (person1);  **// printdata (person2);**  **//invalid(2 different structure variables cant be passed to same function)**  **printdata1(person2); // valid**  return 0;  } |
| **Comparing Structures** To compare structures, you need to compare their fields individually  #include <stdio.h>  #include <string.h>  struct A  {  char name[50];  int age;  } ;  // Function to compare two persons  int comparePersons (struct A p1, struct A p2)  {    if (strcmp(p1.name, p2.name) != 0)  return 0; // Names are different    if (p1.age != p2.age)  return 0; // Ages are different    return 1; // Both are the same  }  int main()  {    struct A person1 = {"Alice", 30};  struct A person2 = {"Alice", 30};  struct A person3 = {"Bob", 35};  if (comparePersons(person1, person2))  printf("Person 1 and Person 2 are the same.\n");  else  printf("Person 1 and Person 2 are different.\n");    if (comparePersons(person1, person3))  printf("Person 1 and Person 3 are the same.\n");  else  printf("Person 1 and Person 3 are different.\n");  }  **Output:**  Person 1 and Person 2 are the same.  Person 1 and Person 3 are different. | |
| **Comparison of variables of 2 different structures is possible**  #include <stdio.h>  #include <string.h>  struct Person1  {  char name[50];  int age;  } ;  struct Person2  {  char name[50];  int age;  } ;  **int comparePersons (struct Person1 p1, struct Person2 p2)**  {    if (strcmp(p1.name, p2.name) != 0)  return 0; // Names are different    if (p1.age != p2.age)  return 0; // Ages are different    return 1; // Both are the same  }  int main()  {    struct Person1 pp1 = {"Alice", 30};  struct Person2 pp2 = {"Alice", 30};    **if (comparePersons(pp1, pp2))**  printf("Person 1 and Person 2 are the same.\n");  else  printf("Person 1 and Person 2 are different.\n");  } | |
| **Array of structures**  #include <stdio.h>  #include <string.h>  struct A  {  char name[50];  int age;  };  void printPersons(struct A persons[], int size)  {  for (int i = 0; i < size; i++)  {  printf("Person %d:\n", i + 1);  printf("Name: %s\n", persons[i].name);  printf("Age: %d\n\n", persons[i].age);  }  }  int main()  {  struct A persons[3];  strcpy(persons[0].name, "Alice");  persons[0].age = 30;  strcpy(persons[1].name, "Bob");  persons[1].age = 25;  strcpy(persons[2].name, "Charlie");  persons[2].age = 35;  printPersons(persons, 3);  return 0;  }  **Output:**  Person 1:  Name: Alice  Age: 30  Person 2:  Name: Bob  Age: 25  Person 3:  Name: Charlie  Age: 35 | |
| **Structures and functions**  #include <stdio.h>  struct student  {  char name[50];  int age;  };  struct student getInformation()  {  struct student s1;  printf("Enter name: ");  scanf ("%s", s1.name);  printf("Enter age: ");  scanf("%d", &s1.age);    return s1;  }  int main()  {  struct student s;  s = getInformation();  printf("\nDisplaying information\n");  printf("Name: %s", s.name);  printf("\nRoll: %d", s.age);    return 0;  } | |
| #include <stdio.h>  struct student  {  char name[50];  int per,rno;  };  int main()  {  struct student s1;  printf("Enter name: ");  gets(s1.name);  printf("Enter the roll number: ");  scanf("%d",&s1.rno);  printf("Enter percentage: ");  scanf("%d", &s1.per);   |  |  | | --- | --- | | display(s1); | display(s1.rno,s1.per); |   return 0;  }   |  |  | | --- | --- | | void display(struct student x )  {  printf("\nDisplaying information\n");  printf("Roll number: %s", x.name);  printf("\nPercentage: %d", x.rno);  } | void display(int a, int b )  {  printf("\nDisplaying information\n");  printf("Roll number: %d", a);  printf("\nPercentage: %d", b);  } | | |
| How to Return Structure from a Function? #include<stdio.h>  struct wage  {  char name[50];  int rs;  };    struct wage employee()  {  struct wage e1;  printf("Enter the name of the employee : ");  scanf("%s",e1.name);  printf("\nEnter the wage : ");  scanf("%d",&e1.rs);  return e1;  }  int main()  {  struct wage e;  e = employee();  printf("\nWage details of the employee\n");  printf("Name : %s",e.name);  printf("\nWage : %d",e.rs);  return 0;  } | |
| **How to Pass Structure by Reference**  #include<stdio.h>  struct car  {  char name[20];  int seat;  char fuel[10];  };  void print\_struct(struct car \*);  int main()  {  struct car tata;  printf("Enter the model name : ");  scanf("%s",tata.name);  printf("\nEnter the seating capacity : ");  scanf("%d",&tata.seat);  printf("\nEnter the fuel type : ");  scanf("%s",tata.fuel);  print\_struct(&tata);  return 0;  }  void print\_struct(struct car \*ptr)  {  printf("\n---Details---\n");  printf("Name: %s\n", ptr->name);  printf("Seat: %d\n", ptr->seat);  printf("Fuel type: %s\n", ptr->fuel);  printf("\n");  } | |
| **An Array of Structures as Function Arguments**  #include<stdio.h>  struct details  {  char name[20];  char sec[20];  float percent;  };  void print\_struct(struct details str\_arr[])  {  int i;  for (i = 0; i < 3; i++)  {  printf("Name: %s\n", str\_arr[i].name);  printf("Section: %s\n", str\_arr[i].sec);  printf("Percentage: %.2f\n", str\_arr[i].percent);  printf("\n");  }  }  int main()  {  struct details student[3] =  {  { "Aisiri","A",89.6 },  { "Gourav","B",60.4 },  { "Samuel","C",98.4 },  };  **print\_struct(student);**  return 0;  } | |
| **Structures and functions**  #include <stdio.h>  #include <string.h>  struct A  {  char name[50];  int age;  };  void initPerson(struct A \*p, const char \*name, int age)  {  strcpy(p->name, name);  p->age = age;  }  void printPerson(const struct A \*p)  {  printf("Name: %s\n", p->name);  printf("Age: %d\n", p->age);  }  void updateAge(struct A \*p, int newAge)  {  p->age = newAge;  }  int main()  {    struct A person1;    initPerson(&person1, "Alice", 30);  printf("Person 1 details:\n");  printPerson(&person1);  updateAge(&person1, 35);  printf("Updated Person 1 details:\n");  printPerson(&person1);  return 0;  } | |
| **Output:**  Name: Alice  Age: 30  Updated Person 1 details:  Name: Alice  Age: 35 | |



**Introduction to pointers**

Pointer is a variable which stores the memory address of another variable. Pointers provide a way to work directly with memory and are a powerful feature of the C programming language.

* A pointer provides a way of accessing a variable without referring to the variable directly.
* The mechanism used for this is the address of the variable.
* A program statement can refer to a variable indirectly using the address of the variable.
* A pointer variable holds the memory address of another variable.
* They are called pointers for the simple reason that by storing an address, they ‘point’ to a particular point in memory.
* A pointer points to that variable by holding a copy of its address.
* Because a pointer holds an address rather than a value, it has two parts.
  + The pointer itself holds the address.
  + The address points to a value.

**Benefits of Using Pointers**

* Pointers can be used to:
* call by address, thereby facilitating the changes made to a variable in the called function to become permanently available in the function from where the function is called.
* return more than one value from a function indirectly
* pass arrays and strings more conveniently from one function to another
* manipulate arrays more easily by moving pointers to them (or to parts of them) instead of moving the arrays themselves
* create complex data structures, such as linked lists and binary trees, where one data structure must contain references to other data structures.
* communicate information about memory, as in the function malloc() which returns the location of free memory by using a pointer.
* compile faster, more efficient code than other derived data types such as arrays.

**Declaration and Initialization of Pointers**

* A pointer has to be declared.
* It will have a value, a scope, a lifetime, a name; and it will occupy a certain number of memory locations.
* The pointer operator available in C is ‘\*’, called value at address operator.
* It returns the value stored at a particular address.
* The value at address operator is also called indirection operator.
* A pointer variable is declared by preceding its name with an asterisk.

**Note:**

1. A pointer should be initialized with another variable’s memory address, with 0, or with the keyword NULL prior to its use; otherwise the result may be a compiler error or a run-time error.
2. Type of the Pointer and the data type of the variable must be same.
3. In C, pointers are not allowed to store any arbitrary memory address, but they can only store addresses of variables of a given type

**Syntax to declare a pointer variable**

**data\_type \*pointer\_name; //\* is called as asterik**

Example: int \* p;

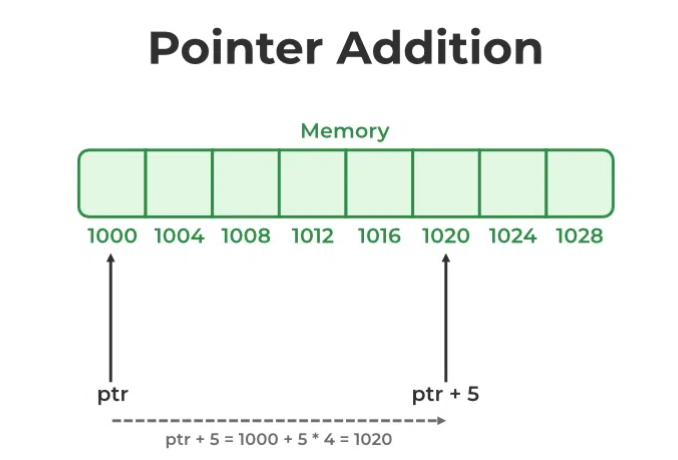
|  |
| --- |
| **Key Differences**  1. **%p**  * Used to print pointers. * %p expects a pointer and prints the address it points to in a platform-dependent format, usually hexadecimal. * %p prints the address in a format that represents the pointer's value, often hexadecimal, with a 0x prefix.  1. **%u**  * Used to print unsigned integers * %u expects an unsigned int and prints its value. * %u prints the number in decimal format. |
| #include <stdio.h>  int main()  {  int i = 5;  int \*p = &i;  printf("\nThe address of i using &num is %p", &i);  printf("\nThe address of i using Ptr is %p", p);  return 0;  }  **Output:**  The address of i using &num is 0x7ffc92f985ac  The address of i using Ptr is 0x7ffc92f985ac   * The placement of the indirection operator before a pointer is said to **dereference the pointer. (\* ptr)** * The value of a dereferenced pointer is not an address, but rather the value at that address— that is, the value of the variable that the pointer points to. |
| #include <stdio.h>  int main()  {  int \*p, c;    c = 22;    p = &c;  printf("1. Address of c: %p\n", p);  printf("2. Content of pointer p: %d\n\n", \*p);    c = 11;    printf("3. Content of pointer pc: %d\n\n", \*p);    printf("4. Address of c: %p\n", p);  printf("5. Address of c: %d\n", p);  printf("6. Address of c: %p\n", &c);    printf("7. %d\n\n", \*p);  printf("8. %d\n\n", c);    printf("9. %p\n\n", &p);  printf("10. %d\n\n", sizeof(p));  printf("11. %d\n\n", sizeof(c));  }  **Output:**  1. Address of c: 0x7ffd8bda33cc  2. Content of pointer p: 22  3. Content of pointer pc: 11  4. Address of c: 0x7ffd8bda33cc  5. Address of c: -1948634164  6. Address of c: 0x7ffd8bda33cc  7. 11  8. 11  9. 0x7ffd8bda33d0  10. 8  11. 4  The %p is a format specifier used in printf to print the memory address of a pointer or a variable. It is specifically designed for printing addresses in hexadecimal format.  The key difference is that c is an integer variable that stores an integer value, while p is a pointer variable that stores a memory address. The memory required to store a memory address (pointer) is generally larger than the memory required to store an integer value. |
| **Example of incorrect usage of pointers by assigning an arbitrary address**  int \*pi;  pi= (int\*)1000; // arbitrary memory address  \*pi = 5; // undefined behaviour, likely crash   In this line, the pointer pi is assigned the value 1000 cast to an int\* (pointer to an integer).   This means pi now holds the memory address 1000. The cast (int\*) tells the compiler to treat the numeric constant 1000 as an address pointing to an integer. |

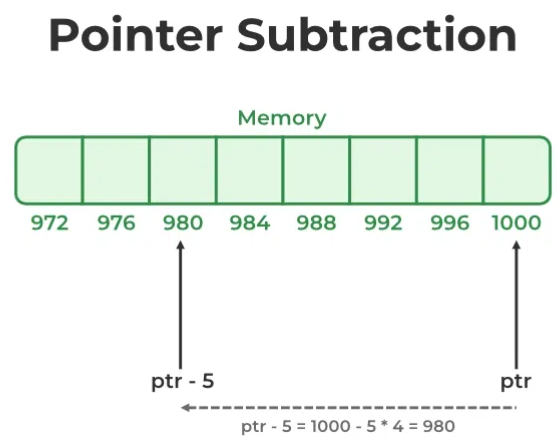
We can perform arithmetic operations to pointer variables using arithmetic operators. We can add an integer or subtract an integer using a pointer pointing to that integer variable

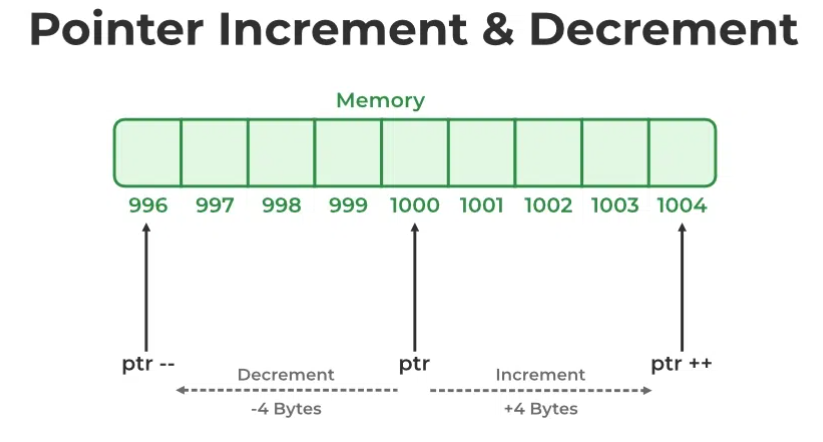
**Note:** While performing division, make sure you put a blank space between ‘/’ and ‘\*’ of the pointer as together it would make a multi-line comment(‘/\*’).

**Example:**

**Incorrect:**  \*ptr\_a/\*ptr\_b;  
 **Correct:**  \*ptr\_a / \*ptr\_b; **google doc, struct memory**  
 **Correct:**  (\*ptr\_a)/(\*ptr\_b);







**NULL POINTERS**

We have seen that a pointer variable is a pointer to some other variable of the same data type. However, in some cases we may prefer to have null pointer which is a special pointer that does not point to any value. This means that a null pointer does not point to any valid memory address. To declare a *null* pointer you may use the predefined constant NULL, which is defined in several standard header files including *<stdio.h>, <stdlib.h>* and *<string.h>*. After including any of these files in your program, write

***int \*ptr = NULL;***

You can always check whether a given pointer variable stores address of some variable or contains a NULL by writing:

***if (ptr = = NULL)***

***{***

***Statement block;***

***}***

You may also initialize a pointer as a null pointer by using a constant 0, as shown below.

int ptr;

ptr = 0;

**Note:** ***It is a logical error to dereference a null pointer.***

|  |  |  |
| --- | --- | --- |
| **Importance of NULL Pointers**  **Indicating No Association**:   * A null pointer is used to signify that a pointer is not currently assigned to any object or memory location. This is especially useful for initializing pointers to ensure they do not contain garbage values that could lead to undefined behaviour if dereferenced.   **Error Handling**:   * Null pointers are often used in error handling and to indicate failure of memory allocation or operations that return pointers. For example, memory allocation functions like malloc return a null pointer when they fail to allocate memory.   **Safe Pointer Checking**:   * Before dereferencing a pointer, it is good practice to check if the pointer is null. This prevents dereferencing null pointers, which can cause program crashes or undefined behaviour.   **Function Arguments and Return Values**:   * Null pointers are often used as sentinel values to indicate that a function did not find a valid result or that an optional argument is not being used.   Sentinel values are special values used in programming to signal certain conditions or to mark the end of a data structure or process. They serve as markers to indicate that processing should stop or that a particular state has been reached. Sentinel values are particularly useful in scenarios such as loops, searches, and data structure traversal.   |  |  | | --- | --- | | if (num == -1) // -1 is the sentinel value  break; | return -1;  // -1 is the sentinel value indicating "not found" |   **End of Data Structures**:   * In data structures like linked lists, trees, and others, null pointers are often used to indicate the end of the structure or absence of a child node.   **Default Values**:   * Null pointers provide a convenient way to initialize pointers with a known value. This helps in writing cleaner code and avoiding uninitialized pointer bugs.  **Memory Allocation:** Using null pointers to check memory allocation failure: **Linked List*:*** Using null pointers to denote the end of a linked list:  Null pointers play a crucial role in C programming by providing a way to indicate that a pointer does not point to any valid memory, aiding in error handling, safe pointer usage, function argument and return value management, and as markers in data structures. Proper use of null pointers helps in writing robust and reliable code, preventing common pointer-related errors such as dereferencing invalid pointers. |

|  |
| --- |
| #include <stdio.h>  int main()  {  int N = 4;  int \*ptr1, \*ptr2;  ptr1 = &N;  ptr2 = &N;    printf("Pointer ptr2 before Addition: ");  printf("%p \n", ptr2);    ptr2 = ptr2 + 3;  printf("Pointer ptr2 after Addition: ");  printf("%p \n", ptr2);    return 0;  }  **Output:**  Pointer ptr2 before Addition: 0x7ffe6f1a2044  Pointer ptr2 after Addition: 0x7ffe6f1a2050 |
| **Write a program to print a character. Also print its ASCII value**  #include <stdio.h>  int main()  {  int ch, \*pch = &ch; // char ch also works  printf("\n Enter the character: ");  scanf("%c", &ch);  printf("\n The char entered is: %c", \*pch);  printf("\n ASCII value of the char is: %d", \*pch);  return 0;  }  **Output:**  Enter the character: A  The char entered is: A  ASCII value of the char is: 65 |

|  |  |
| --- | --- |
| **Pointer expressions and pointer arithmetic**  #include <stdio.h>  int main()  {  int num1=2, num2= 3, sum=0, mul=0, div=1;  int \*ptr1, \*ptr2;  ptr1 = &num1;  ptr2 = &num2;  sum = \*ptr1 + \*ptr2;  mul = sum\* \*ptr1;  printf("%d\n", \*ptr1);  printf("%d\n", \*ptr2);  \*ptr2 =\*ptr2 + 1;  printf("%d\n", \*ptr1);  printf("%d\n", \*ptr2);  div = 9 + (\*ptr1 / \*ptr2) - 30;  printf("%d\n",\*ptr2);  printf("sum=%d\n",sum);  printf("mul=%d\n",mul);  printf("div=%d\n",div);  } | |
| **Passing arguments to functions using pointers (Swapping)** | |
| #include <stdio.h>  **void swap(int \*x, int \*y)**  {  int temp;  temp = \*x;  \*x = \*y;  \*y = temp;  }  main()  {  int a = 10, b = 20;  printf("Before swapping: a = %d, b = %d\n", a, b);  **swap(&a, &b);**  printf("After swapping: a = %d, b = %d\n", a, b);  } | #include <stdio.h>  **void swap(int \*x, int \*y)**  {  int temp;  temp = \*x;  \*x = \*y;  \*y = temp;  }  main()  {  int a = 10, b = 20;int \*p1=&a, \*p2=&b;  printf("Before swapping: a = %d, b = %d\n", a, b);  **swap(p1,p2);**  printf("After swapping: a = %d, b = %d\n", a, b);  } |
| If **swap(\*p1,\*p2) then void swap(int x, int y) and x, y must be returned from swap() to main().** | |

|  |
| --- |
| **C program to demonstrate the use of array of pointers**  #include <stdio.h>  int main()  {    int var1 = 10;  int var2 = 20;  int var3 = 30;  int \*ptr\_arr[3] = { &var1, &var2, &var3 };    for (int i = 0; i < 3; i++)  printf("Value of var **%d**: **%d**\t Address: **%p**\n", **i + 1**, **\*ptr\_arr[i],** **ptr\_arr[i]** );  return 0;  }  **Output:**  Value of var1: 10 Address: 0x7fff1249ed50  Value of var2: 20 Address: 0x7fff1249ed54  Value of var3: 30 Address: 0x7fff1249ed58 |
| **Pointer to an array**  #include<stdio.h>  int main()  {  int arr[] = {10, 20, 30, 40, 50};  int **\*ptr = arr; // \*ptr=&arr or \*ptr=&arr[0] are also valid**  printf("Using array indices:\n");  for (int i = 0; i < 5; i++)  {  printf("arr[%d] = %d\n", i, arr[i]);  }  printf("\nUsing pointer arithmetic:\n");  for (int i = 0; i < 5; i++)  printf("\*(ptr + %d) = %d\n", i, \*(ptr + i));  return 0;  }  **Output:**  Using array indices:  arr[0] = 10  arr[1] = 20  arr[2] = 30  arr[3] = 40  arr[4] = 50  Using pointer arithmetic:  \*(ptr + 0) = 10  \*(ptr + 1) = 20  \*(ptr + 2) = 30  \*(ptr + 3) = 40  \*(ptr + 4) = 50 |
| #include <stdio.h>  int sumArray(int \*arr, int size) //int arr [ ]  {  int sum = 0;  for (int i = 0; i < size; i++)  sum += arr[i];    return sum;  }  int main()  {  int arr[] = {10, 20, 30, 40, 50};  int size = sizeof(arr) / sizeof(arr[0]);  int sum = sumArray(arr, size);  printf("Sum of array elements: %d\n", sum);  return 0;  } |
| **Memory allocation using Pointer**  #include <stdio.h>  #include <stdlib.h>  int main()  {  int n;  printf("Enter the number of elements: ");  scanf("%d", &n);  // Allocate memory for n elements  int \*arr = (int \*)malloc(n \* sizeof(int));  if (arr == NULL)  {  printf("Memory allocation failed!\n");  return 1;  }  // Initialize array  for (int i = 0; i < n; i++)  arr[i] = i + 1;  // Print array elements  printf("Array elements: ");  for (int i = 0; i < n; i++)  printf("%d ", arr[i]);    printf("\n");  // Free allocated memory  free(arr);  return 0;  } |
| **Pointer to pointer**  #include <stdio.h>  int main()  {  int x = 10;  // float x=10 is valid because of void\*  // int \*ptr = (int\*)&x; //type casting  int \*ptr = &x;  int \*\*ptr2 = &ptr;  printf("Value of x: %d\n", x);  printf("Value of x using ptr: %d\n", \*ptr);  printf("Value of x using ptr2: %d\n", \*\*ptr2);  printf("Address of x: %p\n", (void\*)&x);  printf("Address stored in ptr: %p\n", (void\*)ptr);  printf("Address stored in ptr2: %p\n", (void\*)ptr2);  return 0;  }  **Void Pointers (void\*)**:   * A void\* (void pointer) is a special type of pointer that can point to any data type. It is a generic pointer. * void\* pointers are often used in functions that need to accept or return pointers of various types without being concerned about the specific data type.   f oat x = 10;  int \*ptr = &x;  int \*\*ptr2 = &ptr; // may not generate an error  **Why it works?**   **Casting**: Explicit casting can bypass type-checking at compile-time, suppressing errors or warnings. However, it does not guarantee correct behavior.   **Undefined Behavior**: The C standard allows this code to compile, but the behavior at runtime is not predictable or reliable. |
| **Pointer to Structures** #include <stdio.h>  struct Point  {  int x, y;  };  int main()  {  struct Point p = {10, 20};  struct Point \*ptr = &p;  printf("Point coordinates: (%d, %d)\n", ptr->x, ptr->y);  return 0;  }  **Output:**  Point coordinates: (10, 20) ->The arrow operator dereferences the pointer and accesses the specified member in a single operation.**Pointer to Structure**  * struct Point \*ptr = &p; declares a pointer ptr of type struct Point\* and initializes it to point to the address of the structure variable p.  **Accessing Structure Members via Pointer**  * ptr->x and ptr->y are used to access the members x and y of the structure pointed to by ptr.  **Detailed Explanation of ptr->x and ptr->y**  * ptr->x is shorthand for (\*ptr).x. This dereferences the pointer ptr to access the structure it points to and then accesses the x member of that structure. * ptr->y is shorthand for (\*ptr).y. Similarly, this dereferences the pointer ptr to access the structure it points to and then accesses the y member of that structure. |

|  |  |
| --- | --- |
| #include <stdio.h>  struct Point  {  int x;  int y;  };  void printPoint(struct Point \*p)  {  printf("Point coordinates: (%d, %d)\n", p->x, p->y);  }  int main()  {  struct Point p1 = {10, 20};  printPoint(&p1);  return 0;  } | #include <stdio.h>  struct Point  {  int x;  int y;  };  int main()  {  struct Point p1;  struct Point \*ptr = &p1;  ptr->x = 10;  ptr->y = 20;  printf("x: %d, y: %d\n", ptr->x, ptr->y);  return 0;  } |
| The arrow operator (->) should be used to access members of the structure through a pointer.  If p.x and p.y is used then it generates an error. | |
| **Array of Structures with Pointers**  #include <stdio.h>  struct Student  {  char name[50];  int roll\_no;  float marks;  };  int main()  {  struct Student s[3] =  {  {"Alice", 1, 85.5},  {"Bob", 2, 90.0},  {"Charlie", 3, 78.0}  };  struct Student \*ptr = s; // \*ptr=&s is also valid  for (int i = 0; i < 3; i++)  {  printf("Student **%d:** **%s**, Roll No: **%d**, Marks: **%.2f\**n", **i + 1**, **ptr[i].name**, **ptr[i].roll\_no,** **ptr[i].marks**);    }  ptr[1].marks = 95.0;  printf("Updated Marks for Bob: %.2f\n", ptr[1].marks);  return 0;  }   |  | | --- | | In the above example,  When you use ptr[i], you're accessing the i-th element of an array of structures through a pointer. The expression ptr[i] effectively dereferences the pointer ptr to give you the i-th structure itself. So, ptr[i] is a structure, not a pointer to a structure. Therefore, you can use the . operator to access its members.  for (int i = 0; i < 3; i++)  {  printf("Student %d: %s, Roll No: %d, Marks: %.2f\n", i + 1, (ptr + i)->name, (ptr + i)->roll\_no, (ptr + i)->marks);  }  This also works |   **Output:**  Student 1: Alice, Roll No: 1, Marks: 85.50  Student 2: Bob, Roll No: 2, Marks: 90.00  Student 3: Charlie, Roll No: 3, Marks: 78.00  Updated Marks for Bob: 95.00  There is only one pointer variable ptr, but it is used to create pointers to individual structures (ptr[i]) and individual structure members (ptr[i].marks). Each ptr[i] acts as a pointer to a structure, allowing access to its members.  for (int i = 0; i < 3; i++)  {  printf("Student %d: %s, Roll No: %d, Marks: %.2f\n", i + 1, ptr[i].name, ptr[i].roll\_no, ptr[i].marks);  }  The above snippet works even without using the arrow operator (->) because **ptr is an array of structures**, not a single pointer to a structure. | |