MA517M-Basic Programming Laboratory

Laboratory 5 : Pointers

Panchatcharam Mariappan[1]

[1]Associate Professor
Department of Mathematics and Statistics
IIT Tirupati, Tirupati

September 1, 2025

Pointers

Pointers

1. Variable that stores memory addresses
2. Every variable has a memory location, and every memory location has its address
3. & denotes an address in memory
4. Does not store user-given value, instead stores a valid memory address
5. More efficient in handling arrays and structures
6. Return multiple values from functions
7. Dynamic memory allocation and deallocation

Pointers

1. Allows referring to a function as a parameter to functions
2. Pointer variable is basically the same as the other variable, which can store a piece of data
3. A pointer stores a memory address
4. It must be declared before usage
5. The syntax is to place a * in front of the name.
6. *indicates variable is a pointer

Pointer Declaration

```
1 datatype *PtrVariable;
```

What is Pointer?

1. A Variable whose value is the address of another variable or a computer memory location that has an address and holds content.
2. Pointers are powerful as they allow access to addresses and manage their contents
3. Direct access to the memory location
4. Must declare before you can use it to store any variable address
5. Allows us to indirectly access variables, that is, talk about their address rather than their value
6. Correct usage could help to improve the efficiency and performance of the program

Why Pointers?

1. Understanding of pointers and your ability to use separate you from novice programmer to experienced one
2. The basic concept is simple: variable that stores the address of a memory location
3. However, complicated when we start applying pointer operators and discern their cryptic notations
4. Creating fast and efficient code

Why Pointers?

1. Providing a convenient means for addressing many types of problems
2. Supporting dynamic memory allocation
3. Making expressions compact and succinct
4. Providing the ability to pass data structures by pointer without incurring a large overhead
5. Protecting data passed as a parameter to a function

Challenges with Pointers

1. Pointers are complex to handle
2. Wrong usage could lead to many problems, like memory leaks, overflow, and hacking
3. Although Pointer is a powerful tool, the following problems can occur
4. Accessing arrays and other data structures beyond their bounds
5. Reference local variables after they have gone out of existence
6. Referencing heap-allocated memory after it has been released
7. Dereferencing a pointer before memory has been allocated to it

Challenges with Pointers

1. Each pointer must be declared with * prefixed to the name
2. Should be initialized when they are defined or they can be assigned a value
3. A pointer may be initialized to NULL, 0 or an address
4. NULL - A symbolic constant
5. Initializing to 0 is equivalent to initializing NULL

Examples

Examples

```
1     #include<iostream>
2     using namespace std;
3     int main()
4     {
5       int *ptr;
6       double *dptr;
7             return 0;
8     }
```

Examples

```
#include<iostream>
using namespace std;
int main()
{
        int y=5;
        int *yPtr;
        yPtr=&y;
  return 0;
}
```

Address Operator

Address Operator

1. Unary Operator
2. Returns the address of its operand

```
#include<iostream>
using namespace std;
int main()
{
        int y=5;
        int *yPtr;
        yPtr=&y;
  return 0;
}
```

Address Operator

1. The last statement assigns the address of the variable `y` to the pointer variable `yPtr`.
2. `yPtr` is said to point to `y`
3. The operand of the address operator must be a variable
4. The address operator can't be applied to constants or expressions

```
#include<iostream>
using namespace std;
int main()
{
        int y=5;
        int *yPtr;
        yPtr=&y;
  return 0;
}
```

Why Address (&) Operator

1. While declaring the pointers variable, its contents are not initialized
2. It contains some address of somewhere, which is dangerous
3. Should initialize a pointer by assigning a valid address
4. & Operator is used to assign a valid address

Indirection or Dereferencing Operator

1. It operates on a pointer
2. Returns the value stored in the address kept in the pointer variable
3. & and * operator complements each other.

Examples

```
1 int y= 5;
2 int *yPtr;
3 yPtr = &y;  // Declare and assign the address of variable y to yPtr (0x22abfde)
4 cout << yPtr << endl;   // Prints the content of the pointer variable, which contains an address
        (0x22abfde)
5 cout << *yPtr << endl; // Print the value "pointed to" by the pointer, which is an int (5)
6 *yPtr = 8;             // Assign a value to where the pointer is pointed to, NOT to the pointer
      variable
7 cout << *yPtr << endl; // Print the new value "pointed to" by the pointer (8)
8 cout << y << endl;   // The value of variable number changes as well (8)
```

Arrays and Pointers

Arrays and Pointers

1. In C++, an array's name is a pointer, pointing to the first element.
2. Array is treated as a pointer

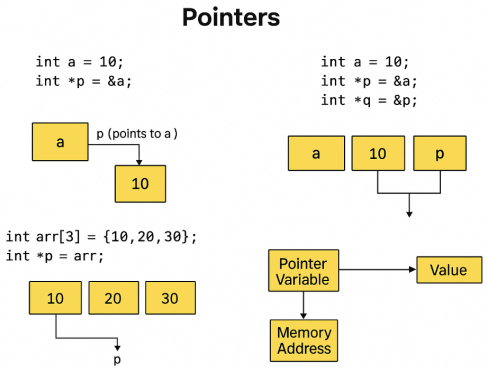

Figure 1: Pointers

Examples

```
1 int x[5]={3,4,5,6,7};‘
2 cout <<x << endl; //shows some address (0x22dacef)
3 cout << *x << endl; //prints 3
4 cout << *(x+1) <<endl; //prints 4
5 cout << *(x+3) << endl; //prints 6
```

In principle, `a[i]` (ith element) can be written as `*(a+i)`

`new` and `delete` operators

`new` Operator

- It is useful to dynamically allocate the storage size at runtime
- When you want to use pointers in place of arrays, it is necessary to allocate the memory
- For example, in VectorAddition.hpp, you may not know the exact size of the vector a priori So, you can allocate while running the program

`new` Operator

- It is useful to dynamically allocate the storage size at runtime
- When you want to use pointers in place of arrays, it is necessary to allocate the memory
- For example, in VectorAddition.hpp, you may not know the exact size of the vector a priori So, you can allocate while running the program

```
1 float *x; //declaration
2 cin>>length; //getting size of vector
3 x = new float[length]; //allocating the spaces for x
```

`new` Operator

1. In our earlier program, we have allocated 50 storage spaces about 200 bytes. But, we have used only 5 values, therefore, 180 bytes are allocated without any usage
2. With the help of pointers, it won't occupy that much memory as you are going to allocate them at runtime. Since the length of the vector is 5, during the runtime, it will allocate only 20 bytes of memory.

The syntax to follow for `new` operator is

```
datatype *VariableName;
VariableName = new datatype[length]; //allocating the
    spaces for x
```

`delete` **Operator**

Whenever you allocate the memory, it should be deleted either at the end of the program or at the end of the function or whenever it is no longer required

```
1 delete VariableName;
```

Examples

Examples

Write a C++ program to add two vectors (VectorAddition.cpp)

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
  float *x,*y,*res;
  int length;
  cin>>length;
  x=new float[length];
  y=new float[length];
  res=new float[length];
  for(int i=0;i<length;i++) {
        x[i]=0.0+i*0.5;
        y[i]=0.0+i*0.5;
        res[i]=0.0;
        }
  for(int i=0;i<length;i++)
      res[i]=x[i]+y[i];
  for(int i=0;i<length;i++)
      cout<<x[i]<<" + "<<y[i]<<" = "<<res[i]<<endl;
  return 0;
}
```

Examples

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
  float *x,*y,*res;
  int length;
  cin>>length;
  x=new float[length];
  y=new float[length];
  res=new float[length];
  for(int i=0;i<length;i++) {
            *(x+i)=0.0+i*0.5;
            *(y+i)=0.0+i*0.5;
            *(res+i)=0.0;
        }
  for(int i=0;i<length;i++)
      *(res+i)=*(x+i)+*(y+i);
  for(int i=0;i<length;i++)
      cout<<*(x+i)<<" + "<<*(y+i)<<" = "<<*(res+i)<<endl;
        return 0;
}
```

Examples

Write a C++ program to compute the Minor of the matrix and the determinant of the matrix (single pointer)

```
#include<iostream>
using namespace std;
void printMatrix(float *A,int row, int col)
{
   cout<<"The matrix is: "<<endl;
   for(int i=0;i<row;i++)
   {
     for(int j=0;j<col;j++)
     {
       cout<<A[i*row+j]<<"\t";
     }
     cout<<endl;
   }
}
```

Examples

```
void Minor(float *A, float *M,int p,int q,int n)
{
  int s=0;
  for(int i=0;i<n;i++)
  {
    for(int j=0;j<n;j++)
    {
      if(i!=p && j!=q)
      {
        M[s++]=A[i*n+j];
      }
    }
  }
}
```

Examples

```
float Determinant(float *A,int n)
{
  float D=0;
  if(n==1)
  {
      D=A[0];
  }
  else if(n==2)
  {
       D=A[0]*A[3]-A[1]*A[2];
  }
  else
  {
    float *Min;
    Min=new float[(n-1)*(n-1)];

    float sign=1;
    for(int i=0;i<n;i++)
    {
            Minor(A,Min,0,i,n);
            printMatrix(Min,n-1,n-1);
            D+=sign*A[i]*Determinant(Min,n-1);
            sign=-sign;
        }
    delete Min;
        }
  return D;
}
```

Examples

```
int main(void)
{
  int row,col;
  cout<<"Enter the row dimension of the matrix n x n :"<<endl;
  cin>>row;
  cout<<"Enter the column dimension of the matrix n x n :"<<endl;
  cin>>col;
  if(row!=col)  {
        cout<<"Determinant is not possible for rectangular matrix"<<endl;
        exit(0);
        }
  float *A, Det;
  A=new float[row*col];
  cout<<"Enter the Entries of the Matrix"<<endl;
  for(int i=0;i<row;i++)  {
      for(int j=0;j<col;j++)  {
      cout<<"The value of A["<<i<<"]["<<j<<"]\n";
      cin>>A[i*row+j];
            }
        }
  printMatrix(A,row,col);
  Det=Determinant(A,row);
  cout<<"The Determinant of the Matrix A is "<<Det<<endl;
  delete A;
  return 0;
}
```

Thanks

Doubts and Suggestions

panch.m@iittp.ac.in