INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

EEC-101
Programming with C++

Module-3:
Pointers

About Subject

- Aggregate Data-types:
  - Arrays
  - Pointers
  - Structures
  - Dynamic data and Pointers
  - Dynamic arrays

Pointers

- A variable
  - whose value is an address
- What can be at that address?
  - Another variable
  - A function
- Pointers point to variables or functions?
- If x is an integer variable and its value is 10
  then I can declare a pointer that points to it
- To use the data that the pointer is pointing to you must know its type

- Pointers is one of the most powerful features of C++
- Pointers enable programs
  - to simulate call-by-reference
  - create and manipulate dynamic data structures

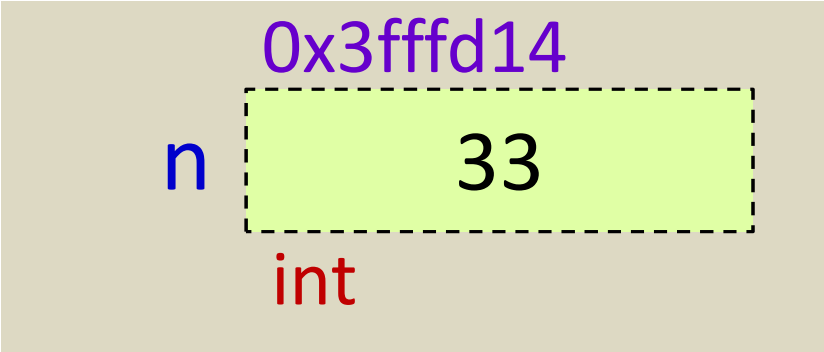
The Reference Operator

- When a variable is declared, three fundamental attributes are associated with it:
  - its *name,*
  - its type, and
  - its *address* in memory.

- For example, the declaration int n;
  - associates the name n
  - the type int
  - the address of some location in memory where the value of n is to be stored.

(suppose that address is 0x3fffd14 in *hexadecimal notation)*

- Visualize *n* like this:



- box represents the variable's storage location in memory
- variable's name is on the left of the box
- variable's type is below the box.
- the variable's address (0x**3fffd14)** is above the box
- If the value of the variable is known (n=33), then it is shown inside the box:

- The value of a variable is accessed by means of its name, e.g, we can print the value of *n* with the statement:

```
cout << n;
```

- The address of a variable is accessed by means of the *address operator &,* e.g, we can print the address of *n* with the statement:

```
cout << &n;
```

- The address operator & “operates” on the variable’s name to produce its address.

Example

```cpp
using namespace std;

int main() {

    int num{100};
    cout << "Value of num is: " << num << endl;
    cout << "sizeof of num is: " << sizeof num << endl;
    cout << "Address of num is: " << &num << endl;
```


```
Value of num is: 100
sizeof of num is: 4
Address of num is: 0x61fe1c
```

Reference

- A reference is an alias, a synonym for another variable.
- It is declared by using the reference operator **&** appended to the reference's type.

```
int n = 55;
int& r = n; // r is a reference for n
```

- Like a constant, a reference must be initialized when it is declared (A synonym must have something for which it is an alias. Every reference must have a referent)

Example

```
using namespace std;
int main() {
    int num{100};
    int &r=num;
    cout<<"num="<<num<<" r="<<r<<endl;
    --num;
    cout<<"num="<<num<<" r="<<r<<endl;
    r*=2;
    cout<<"num="<<num<<" r="<<r<<endl;
    cout<<"address of num="<<&num<<" address of r="<<&r<<endl;
```

```
num=100 r=100
num=99 r=99
num=198 r=198
address of num=0x61fe14 address of r=0x61fe14

Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.
```

Use Cases

- when used as a prefix to a variable name
  - it returns the **address** of that variable (e.g. int n; cout<<&n;)

- when used as a suffix to a type in a variable declaration-
  - it declares the variable to be a synonym for the variable to which it is initialized (e.g. int n = 55; int& r = n;)

- when used as a suffix to a type in a function's parameter declaration
  - it declares the parameter to be a reference parameter for the variable that is passed to it . e.g. swap(int& x, int& y)

- **All of these uses are variations on the same theme: the ampersand refers to the address at which the value is stored.**

Summary

- Reference is another name (alias) for a variable.
- It is a pointer but a constant one – once declared it cannot be made an alias of another variable, e.g , int count; int & refCount = count;
- A reference declaration must have initialization and it can be initialized to a variable.
- A variable can have several references (aliases) – all references hold the same address.
- Reference is not a separate variable– it does not occupy space in memory
- Any arithmetic operation on reference is actually an operation on the referred variable

Pointers

- The reference operator & returns the memory address of the variable to which it is applied.
- We can also store the address in another variable.
- The type of the variable that stores an address is called a ***pointer***.
- If the variable has type int, then the pointer variable must have type “pointer to int ,” denoted by int*

A pointer is a variable and it stores the address of another variable or function.

Why Pointer?

Can't I just use the variable or function itself?

Yes, but not always

- Inside functions, pointers can be used to access data that are defined outside the function. Those variables may not be in scope so you can't access them by their name
- Pointers can be used to operate on arrays very efficiently
- We can allocate memory dynamically on the heap or free store.
  - This memory doesn't even have a variable name.
  - The only way to get to it is via a pointer
- With OO. pointers are how polymorphism works!
- Can access specific addresses in memory
  - useful in embedded and systems applications

Object orientation(OO), or object-oriented programming (OOP), is a programming paradigm that organizes software design around objects, or reusable units of data and behavior.

Pointer

int n; name type memory

- So if I initialize an integer variable named x to 10,i.e,

  int x=10

then x is of type integer and it's bound to some memory location, and it contains the value 10.

- That means that I can declare a pointer variable that stores the address of x.

- So a pointer is a variable. That means that the pointer has a memory location where it's bound to.
- It has a type, it has a value, and the value is an address.

Pointer Declaration

```
variable_type *pointer_name;
```

- we declare pointer variables in exactly the same way except that we add the asterisk prior to the variable name.
- In this context, the asterisk does not function as a mathematical operator, it serves to declare the pointer.
- The way you read these declarations is right to left. So the first example int pointer is a pointer to an integer, double pointer is a pointer to a double.

```
int *int_ptr;      // declares int_ptr to be of type int *// pointer to an integer

int *ptr, n;       // declares ptr  as pointer to int, n is declared to be an integer

double *ptr1, *ptr2; // pointers to double values
```

Pointer Initialization

- Just like all variables, if we don't initialize our variables, they will contain garbage data. In this case, all the pointer variables declared, contain garbage data.

int n;
cout<<n;

- So let's see how we can initialize pointer variables.
- In c++, it's very important that you always initialize all pointer variables before you use them. If you don't initialize a pointer variable, it will have garbage data. In this case, that garbage data represents an address.
- So you can think of an uninitialized pointer as pointing anywhere.
- So, if we use it, we could be accessing memory that we have no business messing around with.
- We don't even know what that memory is.

Null Pointer

Initialize pointers to “point nowhere”.

```
variable_type *pointer_name {nullptr};
```

```
int *int_ptr {};
double* double_ptr {nullptr};
char *char_ptr {nullptr};
string *string_ptr {nullptr};
```

- In these examples, we're initializing the pointer variables to zero that's what null pointer represents this means that we're initializing the pointers to point nowhere.
- We can also initialize pointers to actually point to a variable, and we'll do that next.

- Initializing pointer variables is just like initializing non-pointer variables. We can use an initializer list syntax.

```
int n {0};
```

- Let's review what we just talked about since it's very, very important to understand.
  - Always initialize all pointer variables.
  - Uninitialized pointers contain garbage data and point anywhere.
  - Use null pointer to initialize your pointers unless you initialize them to a variable to a function, this nulls out the pointer.

- One of the most common pointer-related errors that I've seen in code reviews is uninitialized pointers.
- Most of the time this didn't cause a problem, but the potential for trouble is there.

So always initialize your pointers.

Storing an Address in a Pointer Variable

```
int score{10};
double high_temp{100.7};
int * sc_ptr {nullptr};

sc_ptr = &score;
cout << "Value of score is: " << score << endl;
cout << "Address of score is: " << &score << endl;
cout << "Value of score_ptr is: " << sc_ptr << endl;

// score_ptr = &high_temp;      // Compiler error
return 0;
```

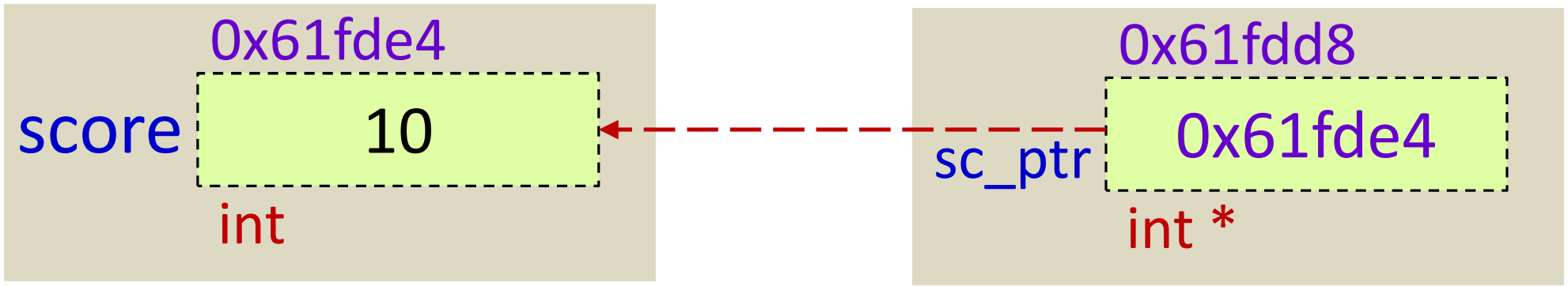
```
Value of score is: 10
Address of score is: 0x61fddc
Value of score_ptr is: 0x61fddc

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

- When we declare pointers, they're typed pointers. This means that we explicitly declare the pointer variable to point to a variable of a specific type.
- In this example, we're declaring an integer score and initializing it to 10 and a double high temp and initializing it to 100.7.
- Then, we declare the score pointer as a pointer to an integer. We assign the address of score to score pointer. The compiler is fine with that since the score pointer holds addresses of integers and scores an integer.
- In the case, when we're assigning the address of high temp to score pointer. The compiler won't let you do this, you'll get a compiler error. The issue is that there's a type conflict.

Example

```
int score{10};
//double high_temp{100.7};
int * sc_ptr {nullptr};

sc_ptr = &score;
cout << "Value of score is: " << score << endl;
cout << "Address of score is: " << &score << endl;
cout << "Value of sc_ptr is: " << sc_ptr << endl;
cout << "Address of sc_ptr is: " << &sc_ptr << endl;
// score_ptr = &high_temp;      // Compiler error
return 0;
```

```
Value of score is: 10
Address of score is: 0x61fde4
Value of sc_ptr is: 0x61fde4
Address of sc_ptr is: 0x61fdd8
```



Accessing Pointer Address

```
int *p;
cout << "\nValue of p is: " << p << endl;    // garbage
cout << "Address of p is: " << &p << endl;
cout << "sizeof of p is: " << sizeof p<< endl;

p = nullptr;
cout << "\nValue of p is: " << p << endl;
```

```
Value of p is: 0x731720
Address of p is: 0x61fe10
sizeof of p is: 8

Value of p is: 0

Process returned 0 (0x0)   execution time : 0.044 s
Press any key to continue.
```

Pointer is a variable.
That means that the pointer has a memory location where it's bound to.
It has a type.
it has a value, and the value is an address.

Size of Pointer Variables

- Don’t Confuse the size of a pointer and the size of what it points to
- All pointers in a program have the same size even though they may be pointing to a very large or very small type.

```
int *p1 {nullptr};
double *p2 {nullptr};
unsigned long long *p3 {nullptr};
vector<string>  *p4{nullptr};
string *p5 {nullptr};

cout << "\nsizeof p1 is: " << sizeof p1 << endl;
cout << "sizeof p2 is: " << sizeof p2 << endl;
cout << "sizeof p3 is: " << sizeof p3 << endl;
cout << "sizeof p4 is: " << sizeof p4 << endl;
cout << "sizeof p5 is: " << sizeof p5 << endl;
```

```
sizeof p1 is: 8
sizeof p2 is: 8
sizeof p3 is: 8
sizeof p4 is: 8
sizeof p5 is: 8
```

Example

- Pointers are variables so they can change
- Pointers can be null
- Pointers can be uninitialized

```
double high_temp{100.7};
double low_temp{100.7};
double * temp_ptr ;                    // points anywhere
temp_ptr =&high_temp;                  // points to high temp
cout<<"temp_ptr="<<temp_ptr<<endl;
temp_ptr =&low_temp;                   // points to high temp
cout<<"temp_ptr="<<temp_ptr<<endl;
temp_ptr =nullptr;                   // points nowhere
cout<<"temp_ptr="<<temp_ptr<<endl;
return 0;
```

```
temp_ptr=0x61fde0
temp_ptr=0x61fdd8
temp_ptr=0

Process returned 0 (0x0)   execution time :
Press any key to continue.
```

Pointers

- The variable p is called a “pointer” because its value “points” to the location of another value.
- If the value to which it points is an int, then it is an int pointer .
- The value of a pointer is an address.
- That address depends upon the state of the individual computer on which the program is running.
- A pointer can be thought of as a “locator”: it tells where to locate another value.

- Often we will need to use the pointer p alone to obtain the value to which it points.

- This is called “**dereferencing**” the pointer and is accomplished simply by applying the star * (the asterisk) symbol as an operator to the pointer (*p)

Dereferencing a Pointer

- Accessing the data we're pointing to is called dereferencing a pointer.

- If sc_ptr is a pointer and has a valid address, then you can access the data at the address contained in the sc_ptr using the dereferencing operator *

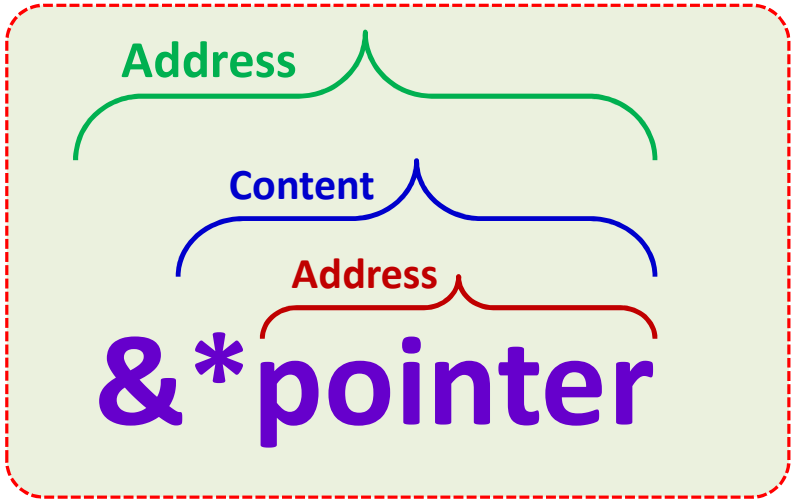
```
int score{10};
int * sc_ptr {nullptr};
sc_ptr = &score;
cout << "Value of score is: " << score << endl;
cout << "Address of score is: " << &score << endl;
cout << "Value of sc_ptr is: " << sc_ptr << endl;
cout << "Address of sc_ptr is: " << &sc_ptr << endl;
cout << "*sc_ptr: " << *sc_ptr << endl;
*sc_ptr*=2;
cout << "*sc_ptr: " << *sc_ptr << endl;
```

```
Value of score is: 10
Address of score is: 0x61fdec
Value of sc_ptr is: 0x61fdec
Address of sc_ptr is: 0x61fde0
*sc_ptr: 10
*sc_ptr: 20
```

Dereferencing

- The address operator & and the dereference operator * are inverses of each other:
- n = = *p whenever p == &n.

- This can also be expressed as n = = *&n and p == &*p.



Example

```
//How to define a pointer
#include <iostream>
using namespace std;
int main()
{
int n = 55;
int* p= &n;                    // p holds the address of n
cout<<"n= "<<n<<endl;
cout <<"&n (address of n) = "<<&n<<endl;
cout<<"p-value of pointer variable p (i.e address of n) = "<<p<<endl;
cout<<"&p- address of pointer varibale="<<&p<<endl;
cout << "&(*p) - address of dereferenced pointer p = "<< & (*p) <<endl;
system ("pause");
return 0;
}
```

```
n= 55
&n (address of n) = 0x61fe1c
p-value of pointer variable p (i.e address of n) = 0x61fe1c
&p- address of pointer varibale=0x61fe10
&(*p) - address of dereferenced pointer p = 0x61fe1c
Press any key to continue . . .
```

Example

```
//The address of a variable is found by using the `&' operator.
//To dereference a pointer (find the value at the address) use the `*' operator

#include <iostream>
using namespace std;

int main()
{
  char ch, *cptr;

  ch = 'A';
  cptr = &ch; // cptr holds address of ch variable

  cout<<"Dereferenced pointer value is        "<< *cptr<<endl; // print 'A'

  *cptr = 'B';

  cout<<"Value in ch variable of char type is "<<ch<<endl;        // print 'B'

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

Summary

- **What Are Pointers?**
- A **pointer** is a variable whose value is the address of another variable.
- Like any variable or constant, you must declare a pointer before you can work with it.
- The general form of a pointer variable declaration is:
- **type *var-name;**

- Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable.
- The asterisk you used to declare a pointer is the same asterisk that you use for multiplication.
- However, in this statement the asterisk is being used to designate a variable as a pointer.

- Pointers are address variables.
- Contain memory address as their values.
- They provide a path to a value via its address rather than its name – indirection

Pointer Arithmetic

- Pointers can be used in
  - Assignment expressions
  - Arithmetic Expressions
  - Comparision Expressions

- Only four arithmetic operators are valid for use with pointers
  - Unary operators:
    - ++(increment) and
    - --(decrement)
  - Binary operators:
    - +(addition) and
    - - (subtraction)

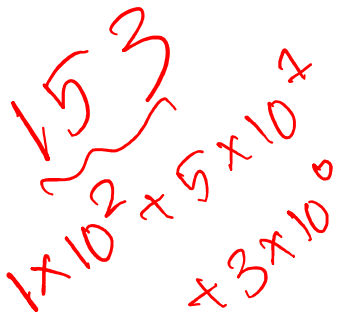
Pointer Arithmetic

- a pointer may be incremented (++) or decremented (--)
- an integer may be added to (+, +=) or may be subtracted from (-, -=) the pointer
- an arithmetic operation moves the pointer forward or backwards. The amount of shift depends on the size of its type

Hexadecimal Numbers

- Decimal number-10 unique symbols (0,1,2,3,4,5,6,7,8,9)
- Hexadecimal number-16 unique symbols (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



Example: Convert $(A7B)_{16}$ to decimal.

$(A7B)_{16} = A \times 16^2 + 7 \times 16^1 + B \times 16^0$

$\Rightarrow (A7B)_{16} = 10 \times 256 + 7 \times 16 + 11 \times 1$ (convert symbols A and B to their decimal equivalents; A = 10, B = 11)

$\Rightarrow (A7B)_{16} = 2560 + 112 + 11$

$\Rightarrow (A7B)_{16} = 2683$

Therefore, the decimal equivalent of $(A7B)_{16}$ is $(2683)_{10}$.

Decimal to Hexadecimal Conversion

$(243)_{10} \longrightarrow (?)_{16}$

16	243	3
	15	

$\longrightarrow (15\ 3)_{16} \longrightarrow (F3)_{16}$

Example: Convert $(92)_{10}$ to hexadecimal.

Solution:

16 | 92

16 | 5 — C

0 — 5

$(5C)_{16}$

$5 \times 16 + C = (92)_{10}$

Hexadecimal Addition

- Basically, hexadecimal addition is similar to decimal addition. But in hexadecimal addition, a carry is generated to the next higher column if the sum is greater than or equal to 16.

Example-1
Add $(5A)_{16}$ and $(BF)_{16}$.

$$\begin{array}{r} 1\ 1 \\ 5\ A \\ +\ B\ F \\ \hline 1\ 1\ 9 \end{array}$$

A	B	C	D	E	F
10	11	12	13	14	15

25 > 16

17 > 16

Example-2
Add $(ABC)_{16}$ and $(2A9)_{16}$.

$$\begin{array}{r} 1\ 1 \\ A\ B\ C \\ 2\ A\ 9 \\ \hline (D\ 6\ 5)_{16} \end{array}$$

$(0000\ 0110\ 1110\ 1111)_2 = (6EF)_{16}$

++ and --

- (++) operator increments a pointer to point to the next element.
- For example, t**he initial memory location of p is 7004** and if the elements that it stores are of two bytes then p++ points next location as 7006

- Similarly (--) decrements the pointer to point to the previous element
- if element is of int type(4 bytes) and initial position is 9004 then, p-- points 9000,

- Also, p=p+2 points to 9012.

+ and -

- `(+)` increment pointer by `n * sizeof(type)`

```
int_ptr += n; or int_ptr = int_ptr + n;
```

- `(-)` decrement pointer by `n * sizeof(type)`

```
int_ptr -= n; or int_ptr = int_ptr - n;
```

if element is of int type(4 bytes) and initial position is 9004 then, int_ptr=int_ptr+3 will move int_ptr to 9004+4*3=9016

Size of Data Types

```cpp
// C++ program to Demonstrate the sizes of data types
#include <iostream>
#include <climits>
using namespace std;
int main()
{

    cout << "Size of short       : " << sizeof(short)<< " bytes" << endl;
    cout << "Size of int         : " << sizeof(int)<< " bytes" << endl;
    cout << "Size of long        : "<< sizeof(long) << " bytes" << endl;
    cout << "Size of float       : "<< sizeof(float) << " bytes" << endl;
    cout << "Size of double      : " << sizeof(double) << " bytes"<< endl;
    cout << "Size of long double : " << sizeof(long double) << " bytes"<< endl;
    return 0;
}
```

```
Size of short       : 2 bytes
Size of int         : 4 bytes
Size of long        : 4 bytes
Size of float       : 4 bytes
Size of double      : 8 bytes
Size of long double : 16 bytes

Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

Example

```
using namespace std;
int main(){
int numi=200;
int* iPtr = &numi;
cout<<"Pointer to integer"<<endl;
cout << iPtr<<endl;
cout << iPtr+1 << endl;
cout << iPtr+2 << endl;
cout << iPtr+3 <<endl<<endl;
double num3 = 55.78623;
double *dPtr = &num3;
cout<<"Pointer to float"<<endl;
cout << dPtr << endl;
cout << dPtr+1 << endl;
cout << dPtr+2 << endl;
cout << dPtr+3 <<endl<<endl;
return 0;
}
```

```
Pointer to integer
0x61fe0c
0x61fe10
0x61fe14
0x61fe18

Pointer to float
0x61fe00
0x61fe08
0x61fe10
0x61fe18


Process returned 0 (0x0)
Press any key to continue.
```

Subtracting two pointers

- Determine the number of elements between the pointers
- Both pointers must point to the same data type

```
int n = int_ptr2 - int_ptr1;
```

Comparing two Pointers

Determine if two pointers point to the same location

- does NOT compare the data where they point!

```
string s1 {"Frank"};
string s2 {"Frank"};

string *p1 {&s1};
string *p2 {&s2};
string *p3 {&s1};

cout << (p1 == p2) << endl;    // false
cout << (p1 == p3) << endl;    // true
```

cout << (*p1 == *p2)

cout << (*p1 == *p3)

Comparing the Data Pointers Point to

Determine if two pointers point to the same data

- you must compare the referenced pointers

```
string s1 {"Frank"};
string s2 {"Frank"};

string *p1 {&s1};
string *p2 {&s2};
string *p3 {&s1};

cout << (*p1 == *p2) << endl;   // true
cout << (*p1 == *p3) << endl;   // true
```

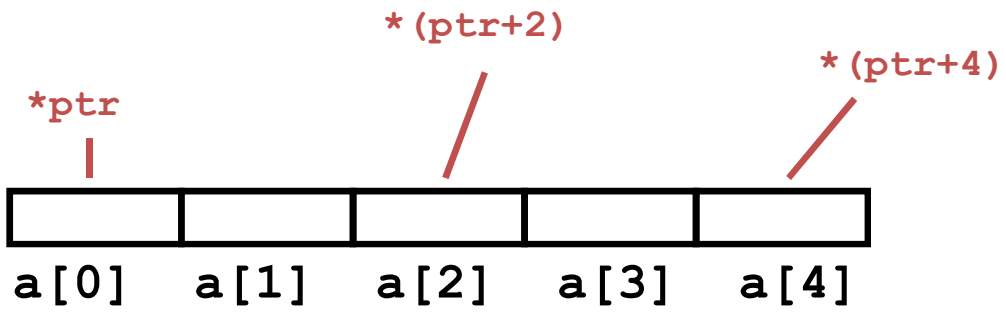
Array and Pointers

- In C++, we can manipulate arrays by using pointers to them. These kinds of pointers that point to the arrays are called array pointers or pointers to arrays.
- a pointer can be made to point to the first element of an array as:

```
int n[5], *nptr;
nptr = n;
// or nptr = &n[0];
```

- nptr can be made to point to any element, for example nptr = &n[2];
- nptr++ will move the pointer one element forward in the array from its current position
- nptr += 2; will make nptr move forward by two elements, i.e., by 8 bytes
- a pointer may be subtracted from another – the operation gives the element-gap between them
- pointers can be compared using equality and relational operators
- **pointer arithmetic is applied mostly on arrays because the elements of an array are stored in contiguous memory**

```
int *ptr = a;
```




```
int a[5];
```

Example

```
#include<iostream>
using namespace std;
int main(){
int arr[]={10, 20,30,40,50 };
for (int* ptr=arr;ptr<arr+10;ptr++)
{
    cout<<"ptr= "<<ptr<<", *ptr= "<<*ptr<<endl;
}
return 0;
}
```

```
ptr= 0x61fe00, *ptr= 10
ptr= 0x61fe04, *ptr= 20
ptr= 0x61fe08, *ptr= 30
ptr= 0x61fe0c, *ptr= 40
ptr= 0x61fe10, *ptr= 50
ptr= 0x61fe14, *ptr= 0
ptr= 0x61fe18, *ptr= 6422040
ptr= 0x61fe1c, *ptr= 0
ptr= 0x61fe20, *ptr= 7804672
ptr= 0x61fe24, *ptr= 0

Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

- Array name is a constant pointer to its first element

```
int main()
{
int mynum[10]= {10,20,30,40,50,60,70,80,90,100};// Declare an array with 10 elements
cout << "Memory address of an array is given by array name      : "<<mynum<<endl<<endl;
cout << "Memory address of the first element of an array        : "<<&mynum[0]<<endl<< endl;
cout<<"*****************************************************"<<endl<< endl;
cout << "Memory address of the fourth element of an array       : "<<&mynum[3]<<endl<< endl;
cout << "Memory address of the fourth element of an array       : "<<mynum+3<<endl<< endl;
cout<<"*****************************************************"<<endl<< endl;

cout << "Value of the first element in the array                :  "<<mynum[0]<<endl<< endl;
cout << "Value of the first element by dereferencing array name :  "<<*mynum<<endl<<endl;
cout<<"*****************************************************"<<endl<< endl;
cout << "Value of the fourth element                            :  "<<mynum[3]<<endl<<endl;
cout << "Value of the fourth element by dereferencing array name:  "<<*(mynum+3)<<endl<<endl;

return 0;
}
```

mynum+3x(4 Byte)

Result

```
Memory address of an array is given by array name      : 0x61fdf0

Memory address of the first element of an array        : 0x61fdf0

************************************************************************

Memory address of the fourth element of an array       : 0x61fdfc

Memory address of the fourth element of an array       : 0x61fdfc

************************************************************************

Value of the first element in the array                :  10

Value of the first element by dereferencing array name :  10

************************************************************************

Value of the fourth element                            :  40

Value of the fourth element by dereferencing array name:  40


Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.
```

Pointers and Arrays

```
int  num[5]; // declare array of 5 integers
```

- **num** is actually a pointer that points to the first element of the array
- Since, the array variable is a pointer, we can dereference it, which returns array element 0

```
int num[5] = { 20, 15, 45, 85, 10 };
// dereferencing an array returns the first element (element 0)
cout << *num; // will displays 20

char ctyName[ ] = “Roorkee”; // C-style string (also an array)
cout << *ctyName; // will display ‘R'
```

Example

```
int main() {
int a[] = { 11, 22, 33, 44 }; // create 4-element array a
int *bPtr = a; // set bPtr to point to array arr

// output array a using array subscript notation
cout<<"Array a printed with: \n\nArray subscript notation\n";
for (int i = 0; i < 4; i++)
cout << "a[" << i << "] = " << a[ i ] << '\n';

 // output array a using the array name and pointer offset notation
cout << "\nPointer/offset notation where "<<"the pointer is the array name\n";
for (int offset1 = 0; offset1 < 4; offset1++)
cout << "*(a + " << offset1 << ") = " << *( a + offset1 ) << '\n';

 // output array b using bPtr and array subscript notation
cout << "\nPointer subscript notation\n";
for (int j = 0; j < 4; j++)
cout << "bptr[" << j << "] = " << bPtr[j] << '\n';
cout << "\nPointer/offset notation\n";
// output array b using bPtr and pointer/offset notation
 for (int offset2 = 0; offset2 <4; offset2++)
cout<<" (bPtr + " << offset2 << ") = "<< * (bPtr + offset2) << '\n';
system ("pause");
```

```
Array a printed with:

Array subscript notation
a[0] = 11
a[1] = 22
a[2] = 33
a[3] = 44

Pointer/offset notation where the pointer is the array name
*(a + 0) = 11
*(a + 1) = 22
*(a + 2) = 33
*(a + 3) = 44

Pointer subscript notation
bptr[0] = 11
bptr[1] = 22
bptr[2] = 33
bptr[3] = 44

Pointer/offset notation
 (bPtr + 0) = 11
 (bPtr + 1) = 22
 (bPtr + 2) = 33
 (bPtr + 3) = 44
Press any key to continue . . .
```

Example

- WAP to find sum of array Elements using Pointer

```
// Sum of array elements using pointer
 #include <iostream>
 using namespace std;
int main() {
int a[15] = { 0 };
int *bPtr = a;
int i;
  for (i=0; i<15;i++)
        a[i]=i;
  cout<<"[a]= [ ";
  for (i=0; i<15;i++)
  cout<<a[i]<<" ";
  cout<<"]"<<endl;
int sum=0;
for (bPtr; bPtr<a+15; bPtr++)
    sum+=*bPtr;
cout<<" sum of elements = "<<sum;
return 0;
}
```

```
[a]= [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ]
 sum of elements = 105
Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

Void Pointer

- NULL and Void pointers are not same.
- A null pointer is a value that any pointer may take to represent that it is pointing to “nowhere”.
- A void pointer is a special type of pointer that can point to somewhere without a specific type.
- NULL pointer refers to the value stored in the pointer itself.
- Void pointer refers to the type of data it points to.

Example: Null Pointer

```
 //NULL pointer
#include <iostream>
using namespace std;
 int main()
{
int num=250;

int* ptr=0;//ptr = NULL;
// *ptr=100; // will generate error as ptr points no where

if (ptr==0) ptr = & num;
cout<<"num="<<*ptr<<endl;
*ptr=100;
cout<<"num="<<num<<endl;
system("pause");
return 0;
}
```

```
num=250
num=100
Press any key to continue . . .
```

Void Pointer

- In C++, a void pointer is a pointer that is declared using the '**void**' keyword (void*).
- It is different from regular pointers it is used to point to data of no specified data type.
- It can point to any type of data so it is also called a "**Generic Pointer**".

- Void Pointer cannot be dereferenced.
- It must be type-casted before dereferencing.
- Arithmetic operations are not allowed on Void Pointers.

Example

```
int main()
{
    int n = 10;
    float f = 25.25;
    char c = '$';
    // Initializing a void pointer
    void* ptr;

    ptr = &n; // pointing to int
    cout << "The value of n :" << n << endl;
    cout << "The Address of n :" << ptr << endl;
    cout << *ptr // error
    ptr = &f; // pointing to float
    cout << "The value of f :" << f << endl;
    cout << "The Address of f :" << ptr << endl;

    ptr = &c; // pointing to char
    cout << "The value of c :" << c << endl;
    cout << "The Address of c :" << ptr << endl;
}
```

```
The value of n :10
The Adress of n :0x61fe14
The value of f :25.25
The Adress of f :0x61fe10
The value of c :$
The Adress of c :0x61fe0f

Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.
```

Example: pointer type-casting

```
using namespace std;
int main()
{
    int n=100;
    // Allocate memory for an integer using new
    void* voidPtr = &n;
    cout<<"address of n= "<<voidPtr;
//    * voidPtr=2000; //will generate error
    // Type casting the void pointer to int* for usage
    int* intPtr = static_cast<int*>(voidPtr);

    // Assign a value to the allocated memory
    *intPtr = 420;
    // Print the value from the allocated memory
    cout <<endl<< "n= " << *intPtr<< endl;
    return 0;
}
```

```
address of n= 0x61fe0c
n= 420
```

Constant Pointers

- In constant pointers, the pointer points to a fixed memory location, and the value at that location can be changed because it is a variable, but the pointer will always point to the same location because it is made constant here.

- A constant pointer is declared as : int *const ptr (the location of 'const' makes the pointer 'ptr' as constant pointer)

Example

```
#include<iostream>
using namespace std;
int main()
{
    int n=100;
    int* const ptr=&n;
    cout<<"value stored in ptr= "<< ptr<<endl;
    cout<<"value stored in the address pointed by ptr= "<< *ptr;
    *ptr=*ptr+100; // value pointed by ptr can change
    cout<<endl<<*ptr;
   // ptr=ptr+1;// error
    return 0;
}
```

```
value stored in ptr= 0x61fe14
value stored in the address pointed by ptr= 100
200
Process returned 0 (0x0)   execution time : 0.026 s
Press any key to continue.
```

Pointer to Constant

- These pointers cannot change the value they are pointing to.
- In other words, they cannot change the value of the variable whose address they are holding.

- Syntax: const int* ptr

- In the pointers to constant, the data pointed by the pointer is constant and cannot be changed. Although, the pointer itself can change and point somewhere else (as the pointer itself is a variable).

Example

```
#include<iostream>
using namespace std;
int main()
{
    int n=100;
    const int*  ptr=&n;
    cout<<"value stored in ptr= "<< ptr<<endl;
    cout<<"value stored in the address pointed by ptr= "<< *ptr;
    //*ptr=*ptr+100; //error: value pointed by ptr can not change
    ptr=ptr+1;// address stored in ptr can change
    cout<<endl<<ptr;
    return 0;
}
```

```
value stored in ptr= 0x61fe14
value stored in the address pointed by ptr= 100
0x61fe18
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Constant Pointer to Constants

- In the constant pointers to constants, the data pointed to by the pointer is constant and cannot be changed.
- The pointer itself is constant and cannot change and point somewhere else.

Example

```
int main()
{
    int a{ 50 };
     int b{ 90 };
      // ptr points to a
     const int* const ptr{ &a };
      //*ptr = 90;
     // Error: assignment of read-only
     // location '*(const int*)ptr'

     // ptr = &b;
     // Error: assignment of read-only
     // variable 'ptr'

     // Address of a
     cout << ptr << "\n";
      // Value of a
```

```
0x61fe0c
50

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

Array of Pointers or Pointer array

- A pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location.
- It is generally used when we want to point at multiple memory locations of a similar data type.
- We can access the data by dereferencing the pointer pointing to it.

Syntax:

```
pointer_type *array_name [array_size];
```

Here,

- **pointer_type:** Type of data the pointer is pointing to.
- **array_name:** Name of the array of pointers.
- **array_size:** Size of the array of pointers.

Array of Pointers

- An array of pointers is an array (stored in contiguous memory locations) of pointer variables. These are also known as pointer arrays.
- The declaration of an array of pointers to an integer –> int *ptr[5];
- This declares ptr as an array of 5 integer pointers. Thus, each element in ptr, now holds a pointer to an int value.

Example: Integer Array

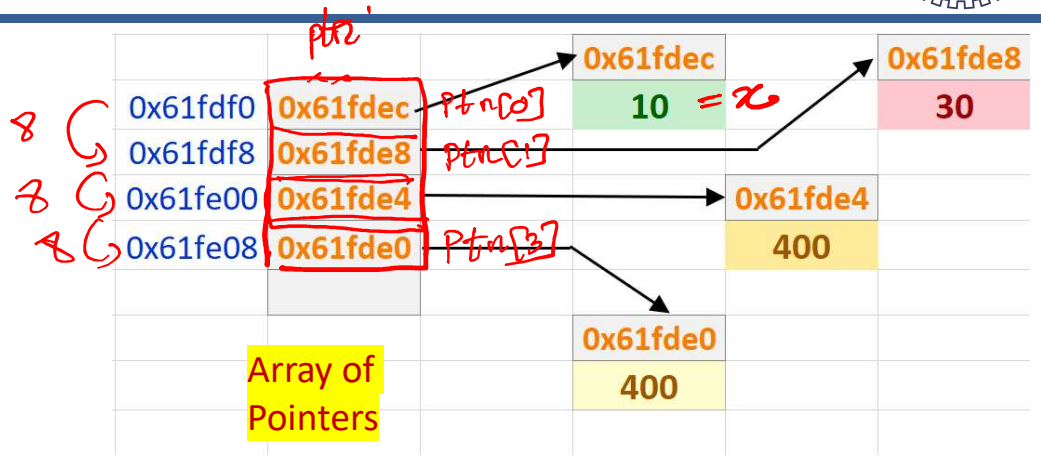
```
#include <iostream>
using namespace std;
const int MAX = 3;
int main () {
   int  var[MAX] = {10, 100, 200};

   for (int i = 0; i < MAX; i++) {

      cout << "Value of var[" << i << "] = ";
      cout << var[i] << endl;
   }
   return 0;
}
```

// array of integers

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200

Process returned 0 (0x0)
Press any key to continue.
```

Example: Array of Pointers

```
#include <iostream>
using namespace std;
const int MAX = 4;
int main () {
    int *ptr[MAX];// declaration of pointer array
    int x=10;
    ptr[0] = &x; // assign the address of integer.
    int y=30;
    ptr[1] = &y;
    int z=400;
    ptr[2] = &z;
    int w=500;
    ptr[3] = &w;
  cout<<"value of x= "<<*ptr[0]<<endl;
    cout<<"value of y= "<<*ptr[1]<<endl;
      cout<<"value of z= "<<*ptr[2]<<endl;
       cout<<"value of w= "<<*ptr[3]<<endl;
for (int i = 0; i < MAX; i++) {
      cout << "Address stored in the pointer array at ptr [" << i << "] = ";
      cout << ptr[i] << endl;
            }
    for (int i = 0; i < MAX; i++) {
      cout << "Address of ptr [" << i << "] = ";
      cout << &ptr[i] << endl;
            }
   return 0;
}
```

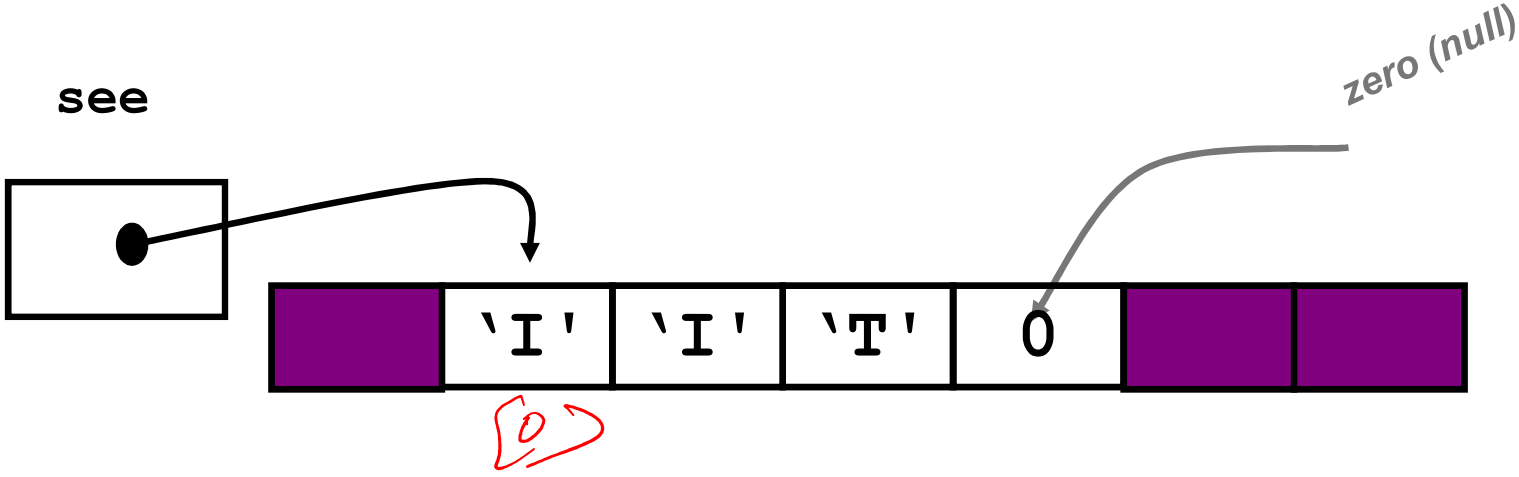



```
value of x= 10
value of y= 30
value of z= 400
value of w= 500
Address stored in the pointer array at ptr [0] = 0x61fdec
Address stored in the pointer array at ptr [1] = 0x61fde8
Address stored in the pointer array at ptr [2] = 0x61fde4
Address stored in the pointer array at ptr [3] = 0x61fde0
Address of ptr [0] = 0x61fdf0
Address of ptr [1] = 0x61fdf8
Address of ptr [2] = 0x61fe00
Address of ptr [3] = 0x61fe08
```

C++ Strings

- A *string* is a *null-terminated* array of characters.
  - null terminated means there is a character at the end of the array that has the value 0 (null).
- Pointers are often used with strings, for example,

```
char *see = "IIT";
```



Array of Pointers

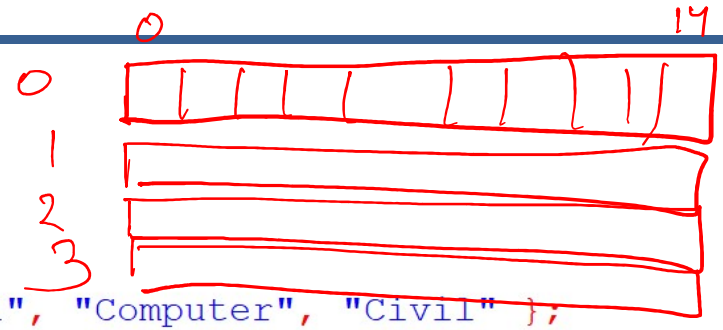
- An array of pointers could be used to create an array of strings (string array).
- In C++, a string is actually a pointer to its first character; thus, a string array would be essentially an array of pointers to the first character of a string in each array's element.

Example: Char Array without pointer

```
#include <iostream>
using namespace std;
int main()
{
const char *branch[4] = { "Architecture", "Electrical", "Computer", "Civil" };

for ( int i=0; i < 6; i++)
    cout << branch[i] << endl;

return 0;
}
```

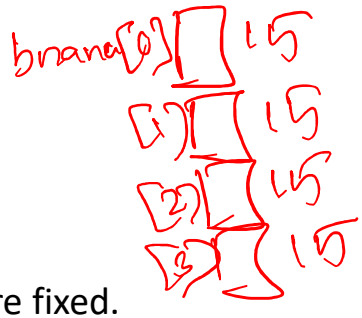
```
Architecture
Electrical
Computer
Civil
í⌠
```

Example



Example: Char Array without pointer

```
#include <iostream>
    using namespace std;
    int main()
    {
char branch[4][15] = { "Architecture", "Electrical", "Computer", "Civil" };

    for ( int i=0; i < 4; i++)
        cout << branch[i] << endl;
    for ( int i=0; i < 4; i++)
        cout << branch[i][1] << endl;
    return 0;
    }
```



```
Architecture
Electrical
Computer
Civil
r
l
o
i
```

- Both the number of strings and the size of the strings are fixed.
- The first dimension (4 in this case), may be left out, and the appropriate size will be computed by the compiler.
- The second dimension, however, must be given (in this case, 15), so that the compiler can choose an appropriate memory layout
- Each string can be modified but will take up the full space given by the second dimension.
- Each will be laid out next to the other in memory and can't change size.

Example

```
#include <iostream>
using namespace std;
const int MAX = 4;
int main () {
char *names[MAX] = { "Rohit Sharma", "Virat Kohli", "Rishabh Pant", "Jasprit Bumrah" };

   for (int i = 0; i < MAX; i++) {
      cout << "Value of names[" << i << "] = ";
      cout << *(names + i)<< " ";
      cout<< "is stored at"<<(names + i)<< endl;
   }

   return 0;
}
```

```
Value of names[0] = Rohit Sharma is stored at0x61fdf0
Value of names[1] = Virat Kohli is stored at0x61fdf8
Value of names[2] = Rishabh Pant is stored at0x61fe00
Value of names[3] = Jasprit Bumrah is stored at0x61fe08

Process returned 0 (0x0)   execution time : 0.053 s
Press any key to continue.
```

Strings Example

- WAP to Count the Number of chars in a string

Method-1

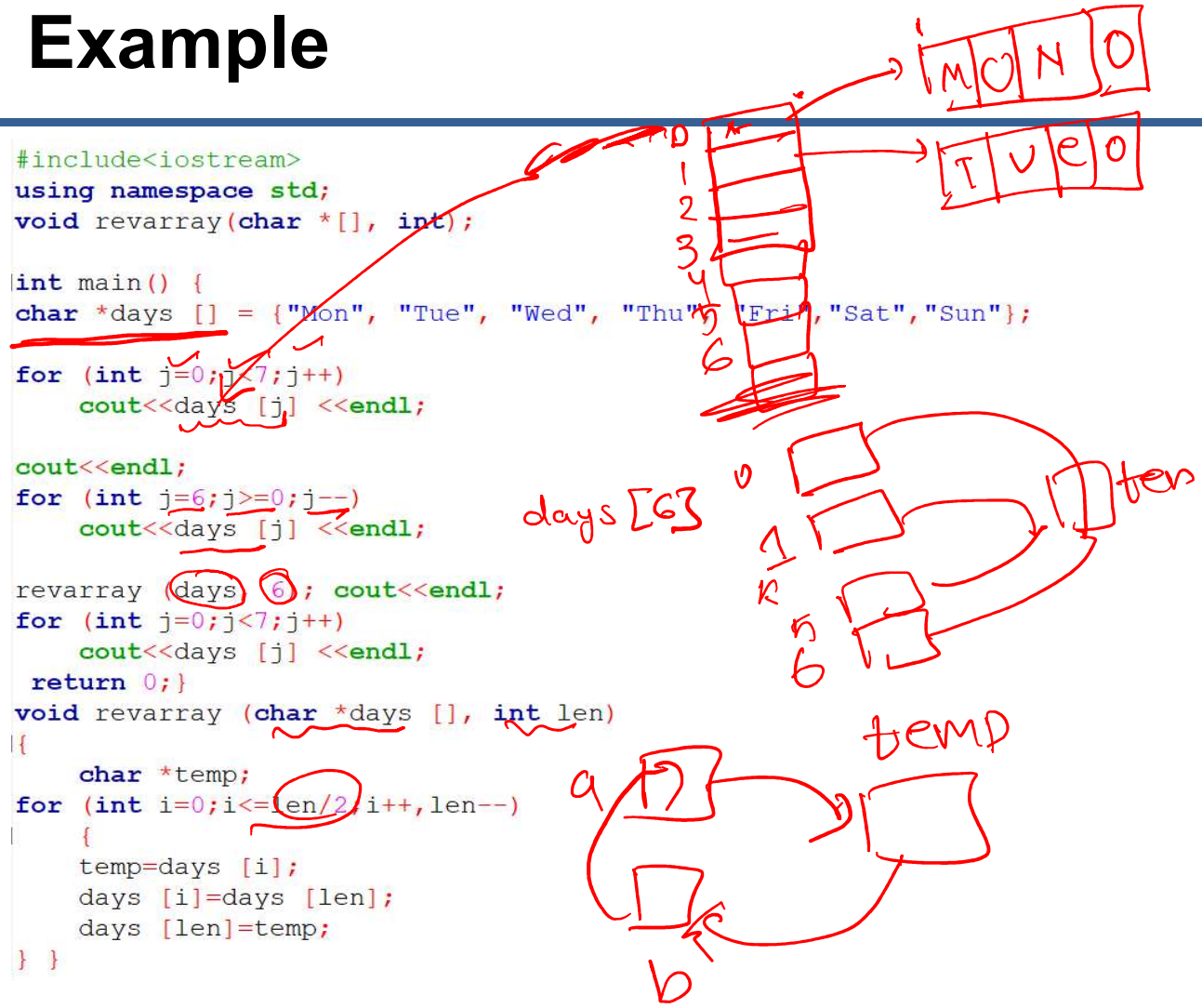
```
int Count( char *s)  //while the thing pointed to by s is not null
{
      int n=0;
      while (*s) {
            n++;    //increment count
            s++;  //set s to point to the next char
      }
      return(n);
}
```

Method-2

```
int count_string( char *s)
 {
 char *ptr = s;
 while (*ptr) {
     ptr++;
 }
 return(ptr - s);
}
```

Example

```
#include<iostream>
using namespace std;
void revarray(char *[], int);

int main() {
char *days [] = {"Mon", "Tue", "Wed", "Thu", "Fri","Sat","Sun"};

for (int j=0;j<7;j++)
    cout<<days [j] <<endl;

cout<<endl;
for (int j=6;j>=0;j--)
    cout<<days [j] <<endl;

revarray (days, 6); cout<<endl;
for (int j=0;j<7;j++)
    cout<<days [j] <<endl;
 return 0;}
void revarray (char *days [], int len)
{
    char *temp;
for (int i=0;i<=len/2;i++,len--)
    {
    temp=days [i];
    days [i]=days [len];
    days [len]=temp;
} }
```




```
Mon
Tue
Wed
Thu
Fri
Sat
Sun

Sun
Sat
Fri
Thu
Wed
Tue
Mon

Sun
Sat
Fri
Thu
Wed
Tue
Mon

Process returned 0 (0x0)   execution time
Press any key to continue.
```

2-D Array Using Pointer

```
//Pointer and Two Dimensional Arrays
#include<iostream>
using namespace std;
int main()
{
int Array[3][3]={{1,2,3}, {4, 5, 6}, {7,8,9}};

for (int i=0;i<3;i++)
{
    for (int j=0;j<3;j++)
{
    cout<<Array[i][j]<<"\t";
}
    cout<<endl;
}
for (int i=0;i<3;i++)
    {
    for (int j=0;j<3;j++)
    {
    cout<<&Array[i][j]<<"\t";
    }
    cout<<endl;
    }
cout<<endl;
int *ptrArr =Array[0];
for (int i=0;i<3;i++)
{
    for (int j=0;j<3;j++)
{
    cout<<*ptrArr++<<"\t";
}
    cout<<endl;
}
system ("pause");
return 0;
}
```

```
1       2       3
4       5       6
7       8       9
0x61fdd0        0x61fdd4        0x61fdd8
0x61fddc        0x61fde0        0x61fde4
0x61fde8        0x61fdec        0x61fdf0

1       2       3
4       5       6
7       8       9
Press any key to continue . . .
```

Pointers and Functions

- Ways to pass arguments to a function
  - by value
  - by reference
  - **by pointer**

- If the function is indented to modify the variables in the calling program:
  - passed by value cannot be used, since the function obtains only the copy of the variable
  - either **a reference argument** or **a pointer can be used**

Passing Pointer to a Function

- Pass-by-reference with pointer parameters
- We can use pointers and the dereference operator to achieve pass-by-reference
- The function parameter is a pointer
- The actual parameter can be a pointer or address of a variable

Pass-by-reference with pointers – defining the function

```
void double_data(int *int_ptr);

void double_data(int *int_ptr) {
   *int_ptr *= 2;

   // *int_ptr = *int_ptr * 2;
}
```

Example

WAP to double a number using pass by reference and also by pointer

```
#include <iostream>
using namespace std;

void doubler (int &a,int &b)
{
 a*=2;
 b*=2;
cout << "Value of 2x= "<<a<<endl;
cout << "Value of 2y= "<<b<<endl;
}
void multby2(int *ptrc,int *ptrd)
{
 *ptrc*=2;
 *ptrd=*ptrd*2;
cout << "Value of 2x= "<<*ptrc<<endl;
cout << "Value of 2y= "<<*ptrd<<endl;
}
 int main ()
 {
    int  x {2}, y{3};

    cout << "Value of x= "<<x<<endl;
    cout << "Value of y= "<<y<<endl;
    // pass by reference
    doubler (x,y);
    // pass by pointer
    multby2 (&x,&y);

    return 0;}
```

```
Value of x= 2
Value of y= 3
Value of 2x= 4
Value of 2y= 6
Value of 2x= 8
Value of 2y= 12

Process returned 0
Press any key to co
```

```
int main ()
{
   int  x {2}, y{3};

   cout << "Value of x= "<<x<<endl;
   cout << "Value of y= "<<y<<endl;
   // pass by reference
   doubler (x,y);
   // pass by pointer
   multby2 (&x,&y);

   return 0;}
```

- When main() function calls the function it supplies the address of the variables as the argument:

  Multby2 (&x, &y);

- These are not the variables that are being passed by reference . They are variables' address.

```
}
void multby2(int *ptrc,int *ptrd)
{
 *ptrc*=2;
 *ptrd=*ptrd*2;
cout << "Value of 2x= "<<*ptrc<<endl;
cout << "Value of 2y= "<<*ptrd<<endl;
}
 int main ()
 {
    int  x {2}, y{3};
```

- Since the multby2( ) function is passed an address, it must use the indirection operator *ptrc, *ptrd to access the value stored at that address.

- For example:

  *ptrx *=2; // (same as *ptrx = *ptrx *2)

```
}
void multby2(int *ptrc,int *ptrd)
{
  *ptrc*=2;
  *ptrd=*ptrd*2;
cout << "Value of 2x= "<<*ptrc<<endl;
cout << "Value of 2y= "<<*ptrd<<endl;
}
  int main ()
{
     int  x {2}, y{3};
```

- ptrc and ptrd contains addresses of x and y respectively
- anything done to *ptrc is actually done to x
- anything done to *ptrd is actually done to y

- Passing a parameter to a function in this case is in some ways similar to passing by reference.
- Both permit the variable in the calling program to be modified by the function

- A reference is an alias for the original value, while a pointer is an address of the variable.

Example

```cpp
using namespace std;

// Function to swap two Numbers using pointers
void swap(int* x, int* y) {
    // Store the value pointed to by x in temp
    int temp = *x;
    // Assign the value pointed by y to the location pointed to by x
    *x = *y;
    // Assign the value stored in temp to the location pointed to by y
    *y = temp;
}
int main() {
    // Initialize two numbers x and y
    int x = 5, y = 10;
    // Print values before swapping
    cout << "Before swapping: x = " << x << ", y = " << y << endl;
    // Call the swap function, passing the addresses of x and y
    swap(&x, &y);
    // Print values after swapping
    cout << "After swapping: x = " << x << ", y = " << y << endl;
    return 0;
}
```

```
Before swapping: x = 5, y = 10
After swapping: x = 10, y = 5
```

Returning a Pointer from a Function

- Functions can also return pointers

```
type *function();
```

- Should return pointers to
  - Memory dynamically allocated in the function
  - To data that was passed in

- Never return a pointer to a local function variable!

Example

```
#include <iostream>
using namespace std;
float * largest (float* ptr1, float*ptr2);

 int main ()
 {
    float x {200.5}, y{30.9};
    float *zptr;
    zptr=largest (&x,&y);
cout<<"largest among "<<x<<" and " <<y<<" = "<<*zptr;
    return 0;}

float * largest (float* ptr1, float*ptr2)
{
if(*ptr1>=*ptr2)
    return ptr1;
else
    return ptr2;
}
```

```
largest among 200.5 and 30.9 = 200.5
Process returned 0 (0x0)   execution time : 0.038 s
Press any key to continue.
```

Never Return a Pointer to a Local variable

```
int *dont_do_this () {
    int size {};
    . . .
    return &size;
}
int *or_this () {
    int size {};
    int *int_ptr {&size};
    . . .
    return int_ptr;
}
```

This will compile just fine since the address of size is the address of an integer and that's what the function returns.

But what's the problem?

- We're returning the address of a local variable in the function.
- The variable's on the stack and the function just terminated, so this variable is now past its lifetime.
- The next time the function is called, or any function is called, the stack area will be reused, and the pointer will now be pointing into that new function's activation record.

- If you overwrite the data it's pointing to, you could trash the
- stack pointers, static links, all kinds of important information on the activation record.

Passing arrays in pointer notation

```
//Pass by pointers
#include <iostream>
using namespace std;
// FUNCTION PROTOTYPE
// passing array to a function by reference using pointers
void twice (int *);
const int arrsize=5;
int main()
{
int myArray [] = { 1, 2, 3, 4, 5};
twice (myArray); // modify elements of iArray (double the value) - iArray is arrays address
for (int i=0; i < arrsize; i++) // display new array elements
cout <<endl<< "myArray["<<i<<"]="<< myArray[i];
cout<<endl<<endl;
system("pause");
return 0;
}
// twice () function header
void twice( int * ptrArr) // ptrarr is equivalent to iArray[]
{
// twice () function definition
// this function multiplies each element by 2

for (int i=0; i < arrsize; i++)
*ptrArr++ *=2; // ptrarr points to elements of iArray Array
}
```

```
myArray[0]=2
myArray[1]=4
myArray[2]=6
myArray[3]=8
myArray[4]=10

Press any key to continue . . .
```

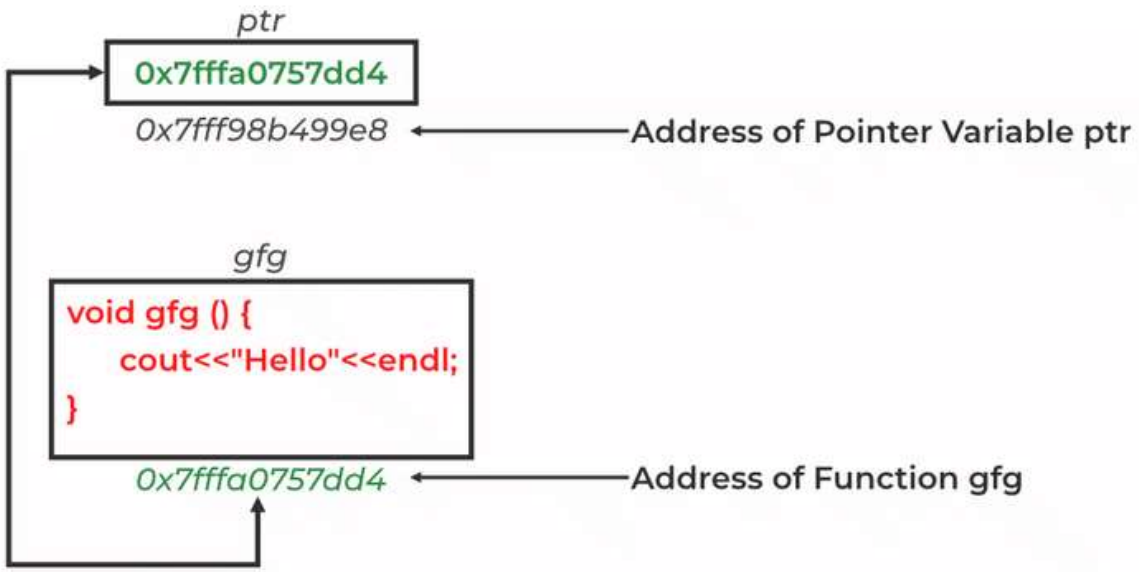
```
// twice () function header
void twice( int * ptrArr) // ptrarr is equivalent to iArray[]
{
// twice () function definition
// this function multiplies each element by 2

for (int i=0; i < arrsize; i++){
//*ptrArr++ *=2; // ptrarr points to elements of iArray Array
*ptrArr=2*(*ptrArr);
ptrArr++ ; }// ptrarr points to elements of iArray Array
}
```

- * ptrArr++ increments the pointer and not the pointer content
- Complier interprets it as *(ptrArr++),i.e., pointer increments first and then indirection or dereference
- *ptrArr++ *=2; // means “
  - increment ptrArr to go to the next element,
  - then dereference its old value (since postfix ++)
  - Similar to writing the following two statements *ptrArr=2*(*ptrArr); ptrArr++ ;

Address of a Function

- **Address of Function:** We all know that every function's code resides in memory, so every function has an address like all other variables in the program.
- The name of a function can be used to find the address of the function.
- We can get the address of a function by just writing the function's name without parentheses.

  - The function pointer is used to point functions, similarly, the pointers are used to point variables.
  - It is utilized to save a function's address.
  - The function pointer is either used to call the function or it can be sent as an argument to another function.


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of Pointer Variable ptr
gfg
void gfg () {
cout<<"Hello"<<endl;
}
0x7fffa0757dd4
Address of Function gfg

Pointer to Functions/Function Pointers

- Like array name, functions name is actually a constant pointer

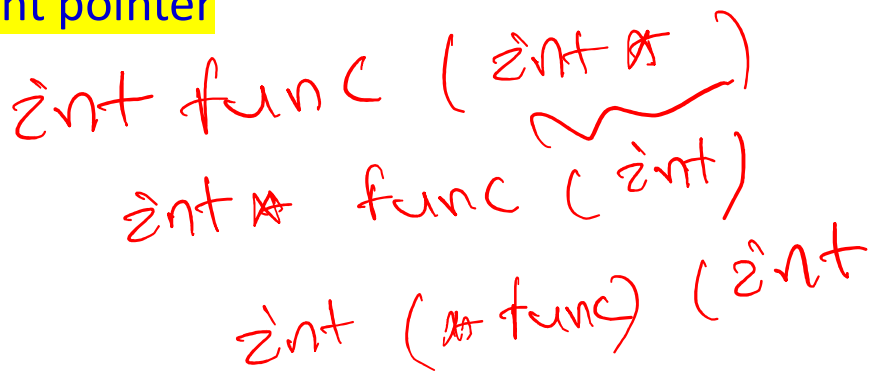
- Visualize value of the pointer as the address of the code that implements the function
- A pointer to a function is a pointer whose value is the address of the name
- A pointer to a function is a pointer to a constant pointer

Let us understand this way:

int func(int); // declares function func

int (*pf)(int x);// declares function pointer pf

pf = &func; OR pf =func; //name function gives its address



Declaration

- Their declaration is easy: write the declaration as it would be for the function, say
  int func (int a, float b);
- and simply put brackets around the name and * in front of it: that declares the pointer.
  int (*func)(int a, float b);

- Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
// function returning pointer to int
 int *func(int a, float b);

//pointer to function returning int
 int (*func)(int a, float b);
```

Calling the Function

- You can call the function using one of two forms:
  - Using Pointer : (*func)(1,2);
  - Using function name: func (1,2);

Example

```
 //C++ program to implementation Function Pointer
#include <iostream>
using namespace std;

int multiply(int a, int b)     {
    cout<<"inside multiply"<<endl;
    return a * b;
    }
int main(){
    int (*func)(int, int);// func is a pointer to a function
    // that takes two integer parameters and return an int
    // it can be any function of this type
    func = multiply;// func is pointing to the multiply

    int z= multiply (10,3);
    cout << "The value of the product is: " << z<< endl;

    int prod = (*func)(15, 2);
    cout << "The value of the product is: " << prod<< endl;

    int mul = func (1, 30);
    cout << "The value of the product is: " << mul<< endl;
    return 0;}
```

```
inside multiply
The value of the product is: 30
inside multiply
The value of the product is: 30
inside multiply
The value of the product is: 30

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

Passing a function pointer as a parameter

- A function pointer stores the memory address of the function.
- When we want to pass the return value to the next function.
- We have two methods to perform this task.
  - First, either pass the value we got
  - Second, pass the function pointer that already exists.

$1^2 + 2^2 + 3^2 + 4^2$

Void Functn_nam (f)

Function

Example

```
// C++ Program for demonstrating
// function pointer as pointer
#include <iostream>
using namespace std;
const int a = 15;
const int b = 2;

// Function for Multiplication
int multiply() {
    return a * b;
    }
// Function containing function pointer
// as parameter
void print(int (*funcptr)())
{
    cout << "The value of the product is: " << funcptr()
         << endl;
}
// Driver Function
int main()
{
    print(multiply);
    return 0;
}
```

```
The value of the product is: 30

Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
```

```
//In this program sum of square values is done using pointer function
#include <iostream>
using namespace std;
int square (int); //function prototype
int (*pf) (int);//pointer to function declaration

 int main()
 {
 pf=&square;
 //pf =square; //pointer referenced to square
int s =0, n=4;
for (int i = 1; i <= n; i++)
 s+=(*pf)(i);//s=s+(*pf)(i)
//s+ square (i); //s=s+square(i)

cout << " sum of square is="<<s<<endl; // 1*1+2+2+3+3+4+4=30 18
return 0;
 }

 int square (int k)
 {
 return k*k;
 }
```

```
 sum of square is=30

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

Example-2

```
//Pointer to functions
#include <iostream>
using namespace std;
// Function prototypes
int sum(int (*) (int),int);// input:(function,int), returns int
int square (int);// input:int, returns int
int cube (int);// input:int, returns int
// main function
int main()
{
cout << sum (square,3) << endl; //1+4+9=14
// call the function sum with square and 3
// note that function parameter is also a functions address
cout << sum (cube,4) << endl; // 1 + 8 +27 + 64=100
system("pause");
return 0;
}
```

```
14
100
Press any key to continue . . .
```

```
int sum(int (*pf) (int k), int n)
{
int s = 0;
for(int i = 1; i <= n; i++)
s+= (*pf)(i);
return s;
}

int square (int k)
{
return k*k;
}
int cube (int k)
{
return k*k*k;
}
```

Array of Function-Pointers

```
//function pointer array
#include <iostream>
using namespace std;
int sum (int, int);
int product (int, int);
int subtract (int, int);
int main()
{
int i = 0;
int a = 10;
int b = 5;
int result = 0;
int (*pfun [3]) (int, int);
pfun [0]= sum;
pfun [1]=product;
pfun [2]=subtract;
for(i=0; i < 3; i++)
{
result = pfun[i] (a, b);
}
cout<<" result="<<result<<endl;
result=pfun [1] (pfun [0] (a, b), pfun [2] (a, b));
//it gives product of sum and subtract
cout<<"The product of the sum and the subtract "<<result<<endl;
 system("pause");
return 0;
}
```

```
int sum (int x, int y){return x+y;}
int subtract (int x, int y){return x-y;}
int product (int x, int y){return x*y;}
```

```
 result=15
 result=50
 result=5
The product of the sum and the subtract 75
Press any key to continue . . .
```

Thanks