Review of C++ Concepts

DSA- Chapter One

Content

- Functions
- Arrays
- Pointers
- dynamic memory allocation
- Structure

1.Function

Functions in C++

- Function is a logically **grouped set of statements that perform a specific task**.
- In C++ program, a function is created to achieve something.
- Every C++ program has at least one function i.e. main() where the execution of the program starts.
- Code within the function starts to execute when the function is called from other functions.

Why functions?

- Avoid repetition of codes.
- Increases program readability.
- Divide a complex problem into simpler ones.
- Reduces chances of error.
- Modifying a program becomes easier by using **function**.

Types of Functions

- Types of Functions in C++, there are two types of functions in C++.
  - **Library Functions**
    - Library functions are **pre-defined functions** in C++.
  - **User-defined Functions**
    - We can also define **our own functions** in C++.

Components of Function

- A function usually has three components. They are:
  - Function Prototype/Declaration
  - Function Definition
  - Function Call

Function Prototype/Declaration

- Function declaration is a statement that informs the compiler about
  - Name of the function
  - Type of arguments
  - Number of arguments
  - Type of Return value

Syntax for function declaration

- We declare a function as follows

  return_type function_name (parameters) ;

  – **return_type:** any valid data type or void (A function with void as return type don't return any value.)
  – **function_name:** any valid cpp identifier
  – **parameters: zero or more** input to the function(separated by comma if more than 1)

- Examples:
  – **float average(int num1, int num2);** /*function name = average, receives two integers as argument and returns float*/
  – int product(int,int); /*function name = product, receives two integers as argument and returns an integer*/

- *Note:*
  – *A function declaration doesn't require name of arguments to be provided, only type of the arguments can be specified.*

Function Definition

- Function definition consists of the body of function.
- The body consists of block of statements that specify what task is to be performed.
- When a function is called, the control is transferred to the function definition.

Syntax for function definition

- Syntax for defining a function is

```
return_type function_name ( [parameters] )
{
   //code
}
```

Example

- Let's see the average function that we defined earlier.

```
float average( int num1, int num2 )
{
   float avg; /* declaring local variable */
   avg = ( num1 + num2 )/2.0;
   return avg; /* returning the average value */
}
```

- Note:
  - While defining functions, it is necessary to specify the parameter type along with the parameter name.
    - Therefore, we wrote 'int' along with num1 and num2.
  - The value the function returns must comply with the return type of the function
    - **return avg;** - returns value of type float.

Calling Function

- A function call can be made by using a call statement.
- A function call statement consists of **function name** and **required argument** enclosed in round brackets.
- To use a function, **we need to call it**.
  - Once we call a function, it performs its operations and after that, the control again passes to the caller function.
- **Syntax:**

  function_name (parameters) ;
- **Example:**
  - If we want to call our average function, we can do it as follows

    average(num1, num2);

Function Call

- A function can be called by two ways. They are:
  - Call by value
  - Call by reference

Call by value

- When a function is called by value, a copy of actual argument is passed to the called function.
- The copied arguments occupy separate memory location than the actual argument.
- If any changes done to those values inside the function, it is only visible inside the function.
- Their values remain unchanged outside it.

Example: call bay value

```cpp
#include <iostream>

float average( int num1, int num2 ); /* declaring function named average */

int main(){
    using namespace std;
    int num1, num2;
    float c;
    cout << "Enter first number" << endl;
    cin >> num1;
    cout << "Enter second number" << endl;
    cin >> num2;
    c = average( num1, num2 ); /* calling the function average and storing its value in c*/
    cout << "Average is " << c << endl;
    return 0;
}

float average( int num1, int num2 ) /* function */
{
        float avg; /* declaring local variable */
        avg = ( num1 + num2 )/2.0;
    return avg; /* returning the average value */
}
```

Call by reference

- In this method of passing parameter, the address of argument is copied instead of value.
- Inside the function, the address of argument is used to access the actual argument.
- If any changes is done to those values inside the function, it is visible both inside and outside the function.

Example: call by reference

```
#include <iostream>
Using namespace std;
void swap(int &, int &); // function prototype
int main() {
   int a,b;
   cout<<"Enter two numbers: ";
   cin>>a>>b;
   cout<<"Before swapping”;;
   cout<<"a =“<<a<<endl;
   cout<<"b =“<<b<<endl;
   swap(a,b); // function call by reference
   cout<<"After swapping\n";
   cout<<"a =“<<a<<endl;
   cout<<"b =“<<b<<endl;    return 0; }
```

```
void swap(int &x, int &y) // function definition
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

Sample Output:

```
Enter two numbers: 12 35
Before swapping
a = 12
b = 35
After swapping
a = 35
b = 12
```

Variable Scope

- **Variable Scope** is a region in a program where a variable is declared and used.
- Depending on the region where variables are declared and used, there are two types of variables
  - **Local** variables
    - Variables that are declared inside a function or a block are called **local variables** and are said to have **local scope**.
    - These local variables can only be used within the function or block in which these are declared.
    - Example: variables in the previous example programs
  - **Global** variables
    - Variables that are defined outside of all the functions and are accessible throughout the program are **global variables** and are said to have **global scope**.
    - Once declared, these can be accessed by any function in the program.

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

// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable
    int global = 2;

    cout << global << endl;
}
```

Global Variable

Local variable

Example:

```
#include <iostream>
using namespace std;
int g = 10;
void func1(){
     g = 20;
     cout << g << endl;
}
int main(){
     func1();
     g = 30;
     cout << g << endl;
     return 0;
}
```

Here, **g** is a **global variable** , since we declared 'g' outside of all the functions and gave it a value in the function.

What is the output?

What if there exists a local variable with the same name as that of global variable inside a function?

```
/*CPP program to illustrate scope of
local variables and global variables
together*/
#include<iostream>
using namespace std;
// global variable
int global = 5;

// main function
int main()
{
   // local variable with same
   // name as that of global variable
   int global = 2;
   cout << global << endl;
}
```

what will be the output? 2 or 5?

- When two variable with same name are defined then the compiler produces a compile time error.
- But if the variables are defined in different scopes then the compiler allows it.
- Whenever there is a local variable defined with same name as that of a global variable then the **compiler will give precedence to the local variable.**

- **Hence, the output is 2**

How to access a global variable when there is a local variable with same name?

```
// C++ program to show that we can access a global
// variable using scope resolution operator :: when
// there is a local variable with same name
#include<iostream>
using namespace std;

int x = 0; // Global x

int main() {
  int x = 10; // Local x
  cout << "Value of global x is " << ::x;
  cout<< "\nValue of local x is " << x;
  return 0;
}
```

Output:
Value of global x is 0
Value of local x is 10

- We use the scope resolution operator (::) to access global variable in the presence of local variable having same name with the global variable
- In C++, scope resolution operator is **::**.
- It is used for the following purposes.
  - **To access a global variable when there is a local variable with same name:**
  - **To define a function outside a class.**
  - **To access a class's static variables.**
  - **For namespace etc**

2.Arrays

Arrays

- An array is a collection of data elements that are of the same type (e.g., a collection of integers, collection of characters, collection of doubles).

Currency Last Trade	U.S. $ N/A	Aust $ Oct 14	U.K. £ Oct 14	Can $ Oct 14	DMark Oct 14	FFranc Oct 14	¥en Oct 14	SFranc Oct 14	Euro 12:39AM
U.S. $	1	0.6493	1.663	0.675	0.5513	0.1644	0.009316	0.6784	1.082
Aust $	1.54	1	2.562	1.04	0.8491	0.2532	0.01435	1.045	1.666
U.K. £	0.6012	0.3904	1	0.4058	0.3314	0.09883	0.005601	0.4079	0.6505
Can $	1.481	0.9619	2.464	1	0.8167	0.2435	0.0138	1.005	1.603
DMark	1.814	1.178	3.017	1.224	1	0.2982	0.0169	1.231	1.963
FFranc	6.083	3.95	10.12	4.106	3.354	1	0.05667	4.127	6.582
¥en	107.3	69.7	178.5	72.46	59.18	17.64	1	72.82	116.1
SFranc	1.474	0.9571	2.452	0.995	0.8126	0.2423	0.01373	1	1.595
Euro	0.9242	0.6001	1.537	0.6238	0.5095	0.1519	0.00861	0.627	1

Arrays

- 1-dimensional array.

Currency Last Trade	U.S. $ N/A	Aust $ Oct 14	U.K. £ Oct 14	Can $ Oct 14	DMark Oct 14	FFranc Oct 14	¥en Oct 14	SFranc Oct 14	Euro 12:39AM
U.S. $	1	0.6493	1.663	0.675	0.5513	0.1644	0.009316	0.6784	1.082

- Two dimensional array

Currency Last Trade	U.S. $ N/A	Aust $ Oct 14	U.K. £ Oct 14	Can $ Oct 14	DMark Oct 14	FFranc Oct 14	¥en Oct 14	SFranc Oct 14	Euro 12:39AM
U.S. $	1	0.6493	1.663	0.675	0.5513	0.1644	0.009316	0.6784	1.082
Aust $	1.54	1	2.562	1.04	0.8491	0.2532	0.01435	1.045	1.666
U.K. £	0.6012	0.3904	1	0.4058	0.3314	0.09883	0.005601	0.4079	0.6505
Can $	1.481	0.9619	2.464	1	0.8167	0.2435	0.0138	1.005	1.603
DMark	1.814	1.178	3.017	1.224	1	0.2982	0.0169	1.231	1.963
FFranc	6.083	3.95	10.12	4.106	3.354	1	0.05667	4.127	6.582
¥en	107.3	69.7	178.5	72.46	59.18	17.64	1	72.82	116.1
SFranc	1.474	0.9571	2.452	0.995	0.8126	0.2423	0.01373	1	1.595
Euro	0.9242	0.6001	1.537	0.6238	0.5095	0.1519	0.00861	0.627	1

- It can also be Multi-dimensional if the data is more than two dimensional

Applications of arrays

- Given a list of test scores, determine the average, maximum and minimum scores.
- Read in a list of student names and rearrange them in alphabetical order (sorting).
- Given the height measurements of students in a class, output the names of those students who are taller than average.

Array Declaration

Currency Last Trade	U.S. $ N/A	Aust $ Oct 14	U.K. £ Oct 14	Can $ Oct 14	DMark Oct 14	FFranc Oct 14	¥en Oct 14	SFranc Oct 14	Euro 12:39AM
U.S. $	1	0.6493	1.663	0.675	0.5513	0.1644	0.009316	0.6784	1.082

- **Syntax:**

  <type> <arrayName>[<array_size>]

  Ex. int Ar[9];

- The array elements are all values of the type **<type>**.
- The size of the array is indicated by **<array_size>**, the number of elements in the array.
- **<array_size>** must be an **int** constant or a constant expression.
- **Note** that an array can have multiple dimensions.

Multi-dimensional array declaration

- **Syntax:**

```
<type> <arrayName>[<num_rows>] >[<num_columns>]
```

```
Ex. int Ar[9][9];
```

Currency Last Trade	U.S. $ N/A	Aust $ Oct 14	U.K. £ Oct 14	Can $ Oct 14	DMark Oct 14	FFranc Oct 14	¥en Oct 14	SFranc Oct 14	Euro 12:39AM
U.S. $	1	0.6493	1.663	0.675	0.5513	0.1644	0.009316	0.6784	1.082
Aust $	1.54	1	2.562	1.04	0.8491	0.2532	0.01435	1.045	1.666
U.K. £	0.6012	0.3904	1	0.4058	0.3314	0.09883	0.005601	0.4079	0.6505
Can $	1.481	0.9619	2.464	1	0.8167	0.2435	0.0138	1.005	1.603
DMark	1.814	1.178	3.017	1.224	1	0.2982	0.0169	1.231	1.963
FFranc	6.083	3.95	10.12	4.106	3.354	1	0.05667	4.127	6.582
¥en	107.3	69.7	178.5	72.46	59.18	17.64	1	72.82	116.1
SFranc	1.474	0.9571	2.452	0.995	0.8126	0.2423	0.01373	1	1.595
Euro	0.9242	0.6001	1.537	0.6238	0.5095	0.1519	0.00861	0.627	1

Accessing Array Elements

- Declare an array of 10 integers:

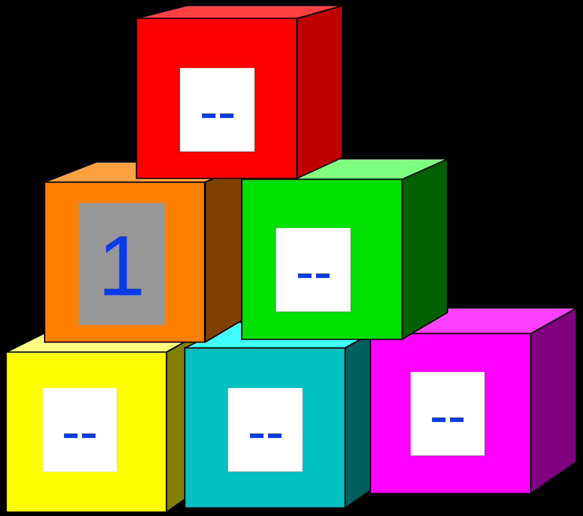
```
int Ar[10];     // array of 10 ints
```

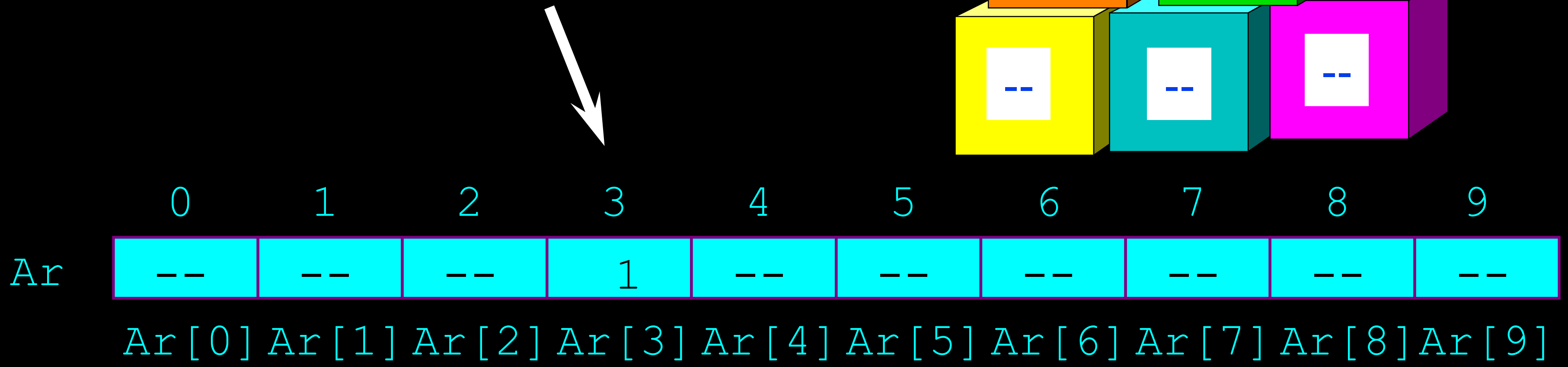
- Subscript(indexing):
  - To access an individual element we must apply a subscript to array named **Ar**.
  - A subscript is a bracketed expression.
    - The expression in the brackets is known as the index.
  - First element of array has index 0. **Ar[0]**
  - Second element of array has index 1, and so on.
    Ar[1], Ar[2], Ar[3],…
  - Last element has an index one less than the size of the array. **Ar[9]**
- Caution: Incorrect indexing is a common error.

Subscripting

```
// array of 10 uninitialized ints
int Ar[10];

Ar[3] = 1;
int x = Ar[3];
```





Array Element Manipulation

- Consider

```
int Ar[10], i = 7, j = 2, k = 4;
Ar[0] = 1;
Ar[i] = 5;
Ar[j] = Ar[i] + 3;
Ar[j+1] = Ar[i] + Ar[0];
Ar[Ar[j]] = 12;
cin >> Ar[k]; // where the next input value is 3
```

	0	1	2	3	4	5	6	7	8	9
Ar	1	--	8	6	3	--	--	5	12	--
	Ar[0]	Ar[1]	Ar[2]	Ar[3]	Ar[4]	Ar[5]	Ar[6]	Ar[7]	Ar[8]	Ar[9]

Array Initialization Ex.

```
int Ar[] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

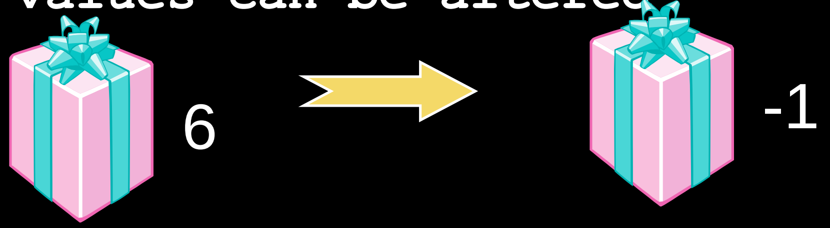
	0	1	2	3	4	5	6	7	8	9
Ar	9	8	7	6	5	4	3	2	1	0

- It can also be initialized like:

```
int Ar[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
```

- Once it is initialized, its values can be altered

```
Ar[3] = -1;
```



	0	1	2	3	4	5	6	7	8	9
Ar	9	8	7	-1	5	4	3	2	1	0

Example: Printing arrays

To print an array, you have to print each element in the array using a loop like the following:

```
for (int i = 0; i < ARRAY_SIZE; i++)
{
  cout << Ar[i] << " ";
}
```

Fill array from keyboard

```
//For loop to fill & print a 10-int array
#include <iostream>
using namespace std;
  int main ( ) {
     int index, ar[10];  // array for 10 integers
      // Read in 10 elements.
     cout << "Enter 10 integers: ";
     for(index = 0; index < 10; index ++)
            cin >> ar[index];
      cout << endl;
      cout << "The integers are ";
      for(index = 0; index < 10; index ++)
            cout << ar[index] << " ";
      cout << endl;
     return 0;
  }
```

Copying Arrays

- Can you copy array using a syntax like this?

  list = myList;

- This is not allowed in C++. You have to copy individual elements from one array to the other as follows:

```
for (int i = 0; i < ARRAY_SIZE; i++)
{
  list[i] = myList[i];
}
```

Summing All Elements

Use a variable named total to store the sum.

Initially total is 0.

Add each element in the array to total using a loop like this:

```
double total = 0;
for (int i = 0; i < ARRAY_SIZE; i++)
{
  total += myList[i];
}
```

Finding the Largest Element

- Use a variable named max to store the largest element.
- Initially max is myList[0].
- To find the largest element in the array myList, compare each element in myList with max, update max if the element is greater than max.

```
double max = myList[0];
for (int i = 1; i < ARRAY_SIZE; i++)
{
  if (myList[i] > max) max = myList[i];
}
```

Finding the index of the largest element

```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < ARRAY_SIZE; i++)
{
 if (myList[i] > max)
 {
  max = myList[i];
  indexOfMax = i;
 }
}
```

Shifting Elements

```
Int myListSize= 10;
double temp = myList[0]; // Retain the first element
// Shift elements left
for (int i = 1; i < myListSize; i++)
{
  myList[i - 1] = myList[i];
}
// Move the first element to fill in the last position
myList[myListSize - 1] = temp;
```

Passing Array to a Function

- In C++, we can pass arrays as an argument to a function. And, also we can return arrays from a function.
- There are two syntax for declaring function with array parameter
  1. returnType functionName(dataType arrayName[ ], int size)

     Example:
     - Int sum(int marks[], int size);
  2. returnType functionName(dataType arrayName[arraySize])

     Example:
     - Int sum(int marks[5]);
- When we call a function by passing an array as the argument, only the name of the array is used.
  – functionName(arrayName); //note there is no[ ] during function call

Example

```
void display(int m[5]) {
    cout << "Displaying marks: " << endl;
    // display array elements
    for (int i = 0; i < 5; ++i)
        cout << "Student " << i + 1 << ": " << m[i] << endl;
}
int main() {
    // declare and initialize an array
    int marks[5] = {88, 76, 90, 61, 69};
    // call display function and pass array as argument with no size and [ ] operator
    display(marks);
    return 0;
}
```

Exercise 1: What is the output?

```cpp
#include <iostream>

using namespace std;

int main()
{
    int n[10]; /* declaring n as an array of 10 integers */
    int i,j;
    /* initializing elements of array n */
     for ( i = 0; i<10; i++ )
     {
        cout << "Enter value of n[" << i << "]"<< endl;
                cin >> n[i];
    }
    /* printing the values of elements of array */
    for (j = 0; j < 10; j++ )
    {
               cout << "n[" << j << "] = " << n[j] << endl;
    }
    return 0;
}
```

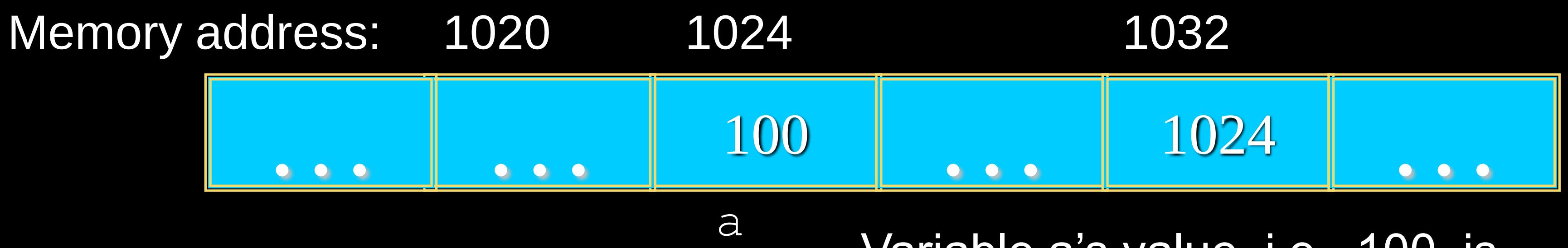
Exercises

- Find the average of an integer array
  - Read 10 integers from the keyboard
  - Calculate and display the average value
- Find second largest element in an given array of integer
- Find the largest three elements in an array

3.Pointers

Computer Memory

- Each variable is assigned a memory slot (the size depends on the data type) and the variable’s data is stored there

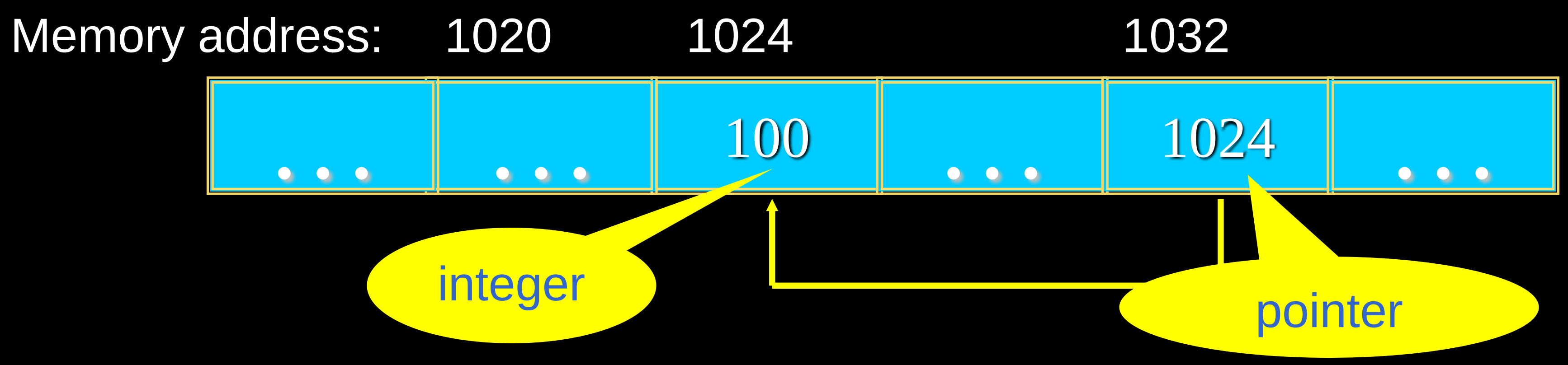



```
int a = 100;
```

Variable a’s value, i.e., 100, is stored at memory location 1024

Pointers

- A pointer is a variable used to store the address of a memory cell.
- We can use the pointer to reference this memory cell



Pointer Types

- Pointer
  - C++ has pointer types for each type of object
    - Pointers to `int` objects
    - Pointers to `char` objects
    - Pointers to user-defined objects
      - Struct type(e.g., `Student`)
      - Class type
  - Even pointers to pointers
    - Pointers to pointers that points to `int` objects

Pointer Variable

- Declaration of Pointer variables

```
type* pointer_name;
 //or
type *pointer_name;
```

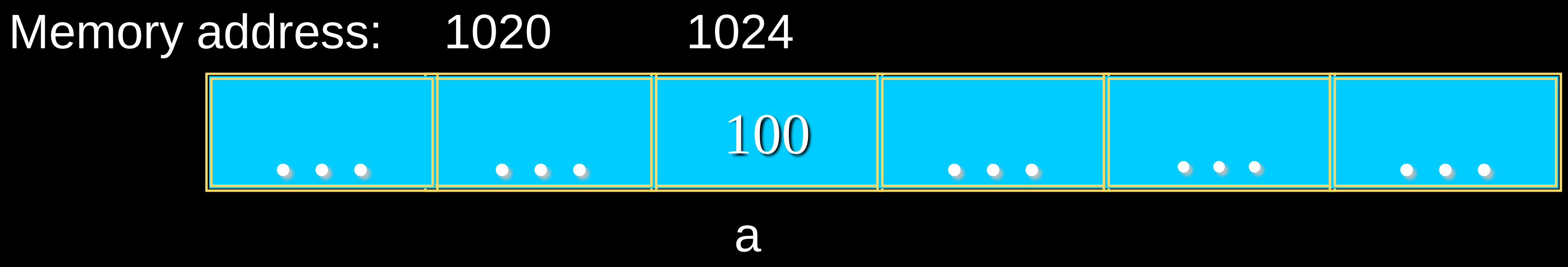
where *type* is the type of data pointed to (e.g. int, char, double)

Examples:

```
int *n;
RationalNumber *r;
int **p;    // pointer to pointer
```

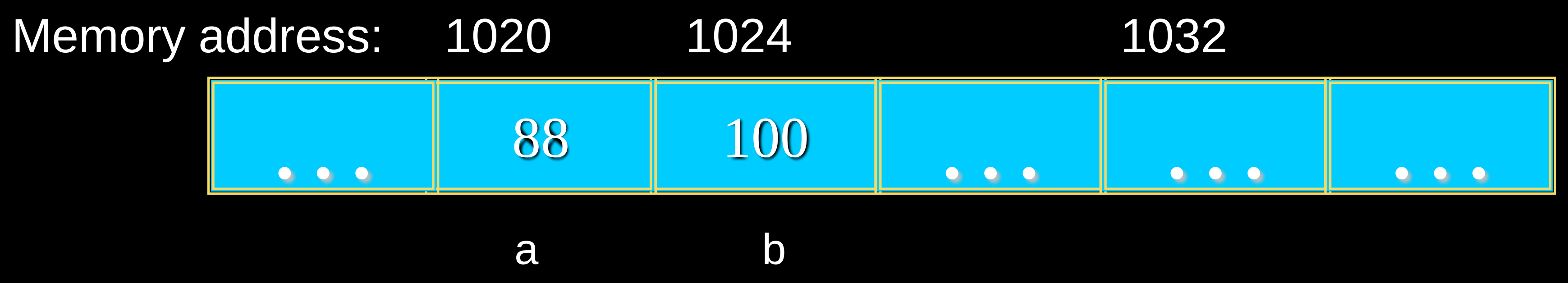
Address Operator &

- *The "address of" operator* (&) gives the memory address of the variable
  - Usage: **&variable_name**




```
int a = 100;
//get the value,
cout << a;     //prints 100
//get the memory address
cout << &a;    //prints 1024
```

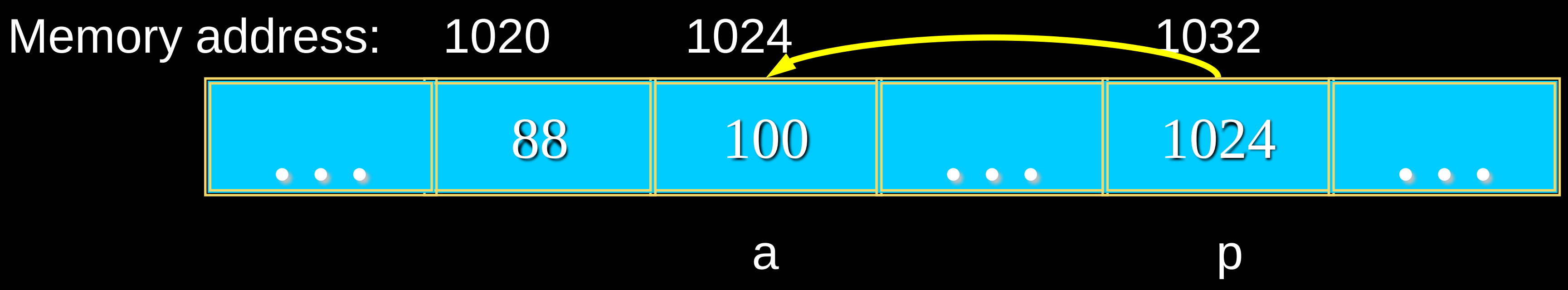
Address Operator &




```
#include <iostream>
using namespace std;
void main(){
   int a, b;
   a = 88;
   b = 100;
   cout << "The address of a is: " << &a << endl;
   cout << "The address of b is: " << &b << endl;
}
```

Result is:
The address of a is: 1020
The address of b is: 1024

Pointer Variables



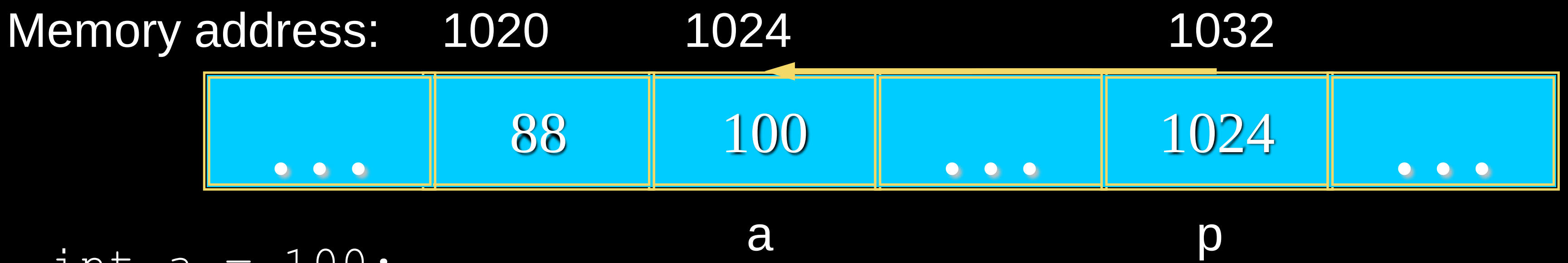

```
int a = 100;
int *p = &a;
cout << a << " " << &a <<endl;
cout << p << " " << &p <<endl;
```

Result is:
100 1024
1024 1032

- The value of pointer `p` is the address of variable `a`
- A pointer is also a variable, so it has its own memory address

Dereferencing Operator *

- We can access to the value stored in the variable pointed to by using the dereferencing operator (*),

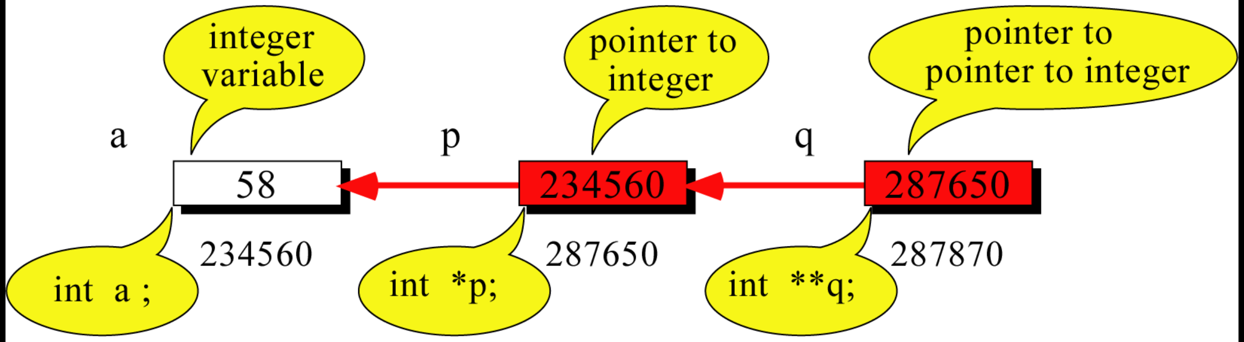



```
int a = 100;
int *p = &a;
cout << a << endl;
cout << &a << endl;
cout << p << " " << *p << endl;
cout << &p << endl;
```

Result is:
100
1024
1024 100
1032

Pointer to Pointer

```
// Local Declarations
int        a ;
int       *p ;
int      **q ;
```




```
// Statements
a = 58 ;
p = &a ;
q = &p ;
cout <<       a << " ";
cout <<      *p << " ";
cout <<     **q << " ";
```

What is the output?

58 58 58

Don’t get confused

- Declaring a pointer means only that it is a pointer:
  - `int *p;`
- Don’t be confused with the dereferencing operator, which is also written with an asterisk (*).
  - They are simply two different tasks represented with the same sign

```
int a = 100, b = 88, c = 8;
int *p1 = &a, *p2, *p3 = &c;
p2 = &b;       // p2 points to b
p2 = p1;       // p2 points to a
b = *p3;       //assign c to b
*p2 = *p3;     //assign c to a
cout << a << b << c;
```

Result is:
888

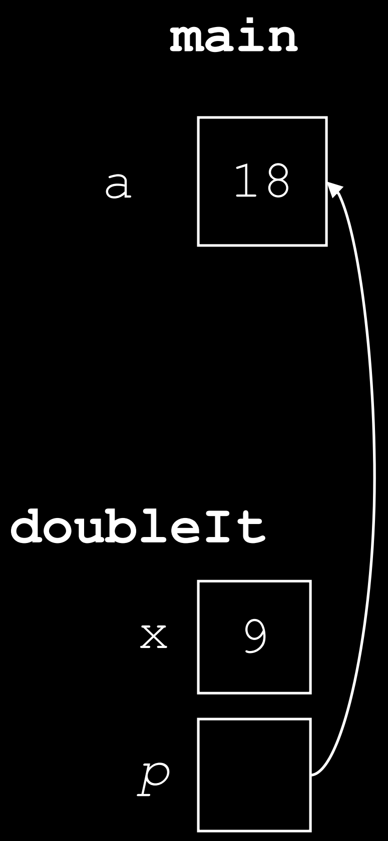
A Pointer Example

The code

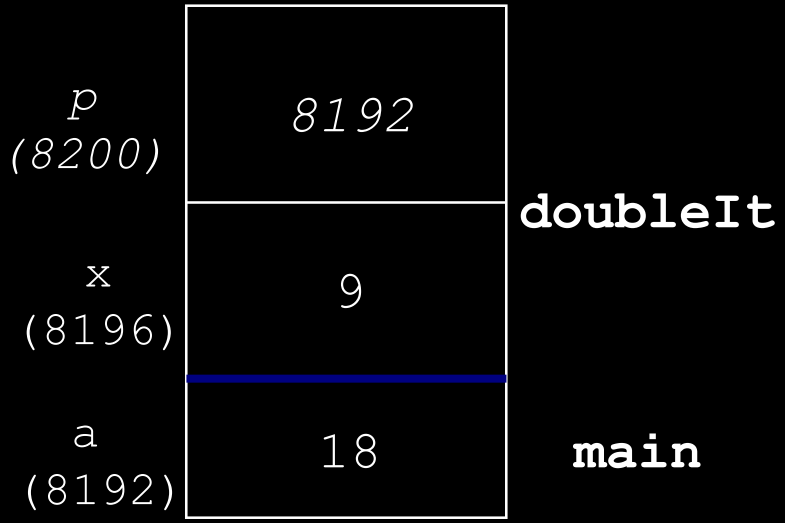
```
void doubleIt(int x, int * p)
{
   *p = 2 * x;
}
int main()
{
   int a = 16;
   doubleIt(9, &a);
   cout<<“a gets “<<a;
   return 0;
}
```

a gets 18

Box diagram



Memory Layout



Another Pointer Example

```
#include <iostream>
using namespace std;
int main (){
   int value1 = 5, value2 = 15;
   int *p1, *p2;
   p1 = &value1; // p1 = address of value1
   p2 = &value2; // p2 = address of value2
   *p1 = 10;     // value pointed to by p1=10
   *p2 = *p1;    // value pointed to by p2= value pointed to by p1
   p1 = p2;          // p1 = p2 (pointer value copied)
   *p1 = 20;     // value pointed to by p1 = 20
   cout << "value1==" << value1 << "/ value2==" << value2;
   return 0;
}
```

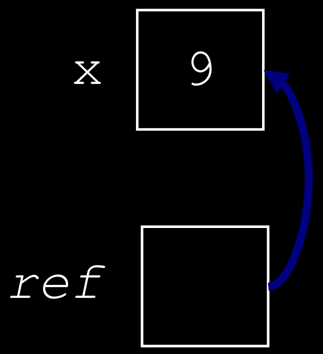
Result is

value1==10 / value2==20

Reference Variables

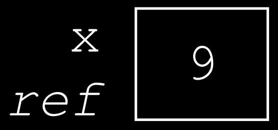
A reference is an additional name to an existing memory location

Pointer:



int x=9;
int *ref;
ref = &x;

Reference:



int x = 9;
int &ref = x;

Reference Variables

- A **reference variable** serves as an alternative name for an object

```
int m = 10;
int &j = m;  // j is a reference variable
cout << "value of m = " << m << endl;
                  //print 10
j = 18;
cout << "value of m = " << m << endl;
     // print 18
```

value of m = 10
value of m = 18

Reference Variables

- A reference variable always refers to the same object.
- Assigning a reference variable with a new value actually changes the value of the referred object.
- Reference variables are commonly used for parameter passing to a function

Traditional Pointer Usage

```
void IndirectSwap(char *Ptr1, char *Ptr2){
  char temp = *Ptr1;
  *Ptr1 = *Ptr2;
  *Ptr2 = temp;
}

int main() {
  char a = 'y';
  char b = 'n';
  IndirectSwap(&a, &b);
  cout << a << b << endl;
  return 0;
}
```

Pass by Reference

```
void IndirectSwap(char& y, char& z) {
  char temp = y;
  y = z;
  z = temp;
}

int main() {
  char a = 'y';
  char b = 'n';
  IndirectSwap(a, b);
  cout << a << b << endl;
  return 0;
}
```

NULL pointer

- NULL is a special value that indicates an empty pointer
- If you try to access a NULL pointer, you will get an error

```
int *p;
p = 0;
cout << p << endl; //prints 0
cout << &p << endl;//prints address of p
cout << *p << endl;//Error!
```

Pointers in Array

- Pointer to the array holds the **address of the first element of the array** i.e., array[0].
- **Similarly, array name is a pointer to the first element of the array.**
- If **p** is a pointer to the array **age**, then it means that p(or age) points to age[0].

```
int age[50];
int *p;
p = age;
```

  - The above code assigns the address of the first element of age to p.
  - Now, since **p** points to the first element of the array **age**, ***p** is the value of the first element of the array.
- **So, *p is age[0], *(p+1) is age[1], *(p+2) is age[2].**
  - Similarly, *age is age[0] (value at age), *(age+1) is age[1] (value at age+1), *(age+2) is age[2] (value at age+2) and so on.

4.Dynamic Memory Allocation

Memory Management

- Static Memory Allocation
  - Memory is allocated at compilation time
- Dynamic Memory
  - Memory is allocated at running time

Static vs. Dynamic Objects

Static object

(variables as declared)

- Memory is acquired automatically

- Memory is returned automatically when object goes out of scope

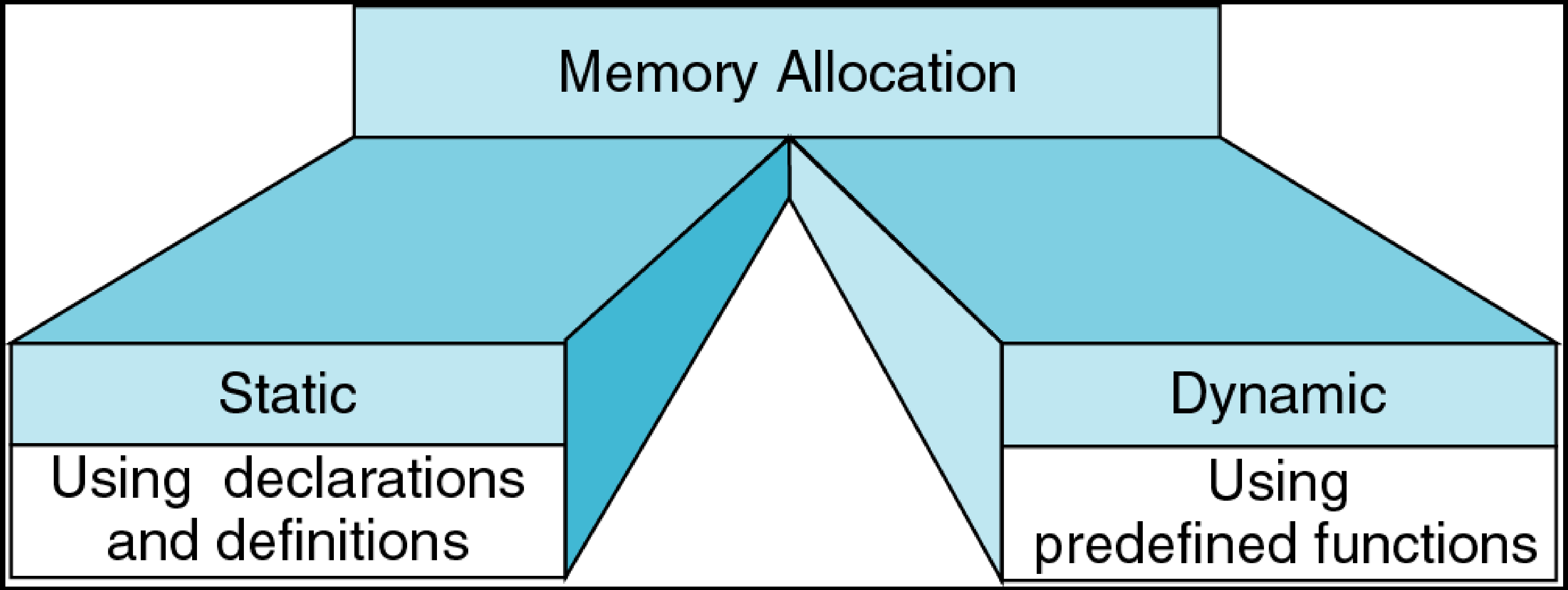
Dynamic object

- Memory is acquired by program with an allocation request
  - **`new`** operation

- Dynamic objects can exist beyond the function in which they were allocated

- Object memory is returned by a de-allocation request
  - **`delete`** operation

Memory Allocation



new
delete

```
{
   int a[200];
   …
}
```

```
int* ptr;
ptr = new int[200];
…
delete [] ptr;
```

Object (variable) creation: `New`

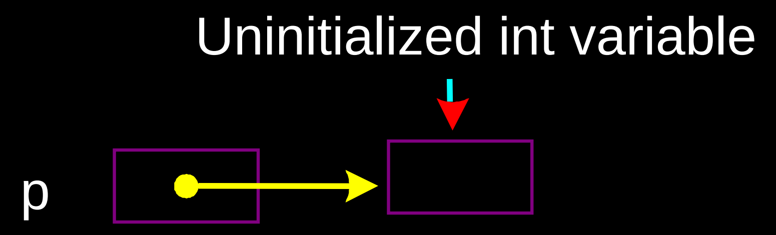
Syntax

```
ptr = new SomeType;
```

where `ptr` is a pointer of type `SomeType`

Example

```
int* p = new int;
```



Object (variable) destruction: Delete

Syntax

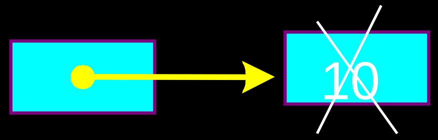
```
delete p;
```

storage pointed to by p is returned to free store and p is now undefined

Example

```
int* p = new int;
*p = 10;
delete p;
```

p



Array of New: **dynamic arrays**

- Syntax

```
SomeType *P = new SomeType[Expression];
```

  - Where
    - `P` is a pointer of type `SomeType`
    - `Expression` is the number of objects to be constructed -- we are making an array

- Because of the flexible pointer syntax, `P` can be considered to be an array

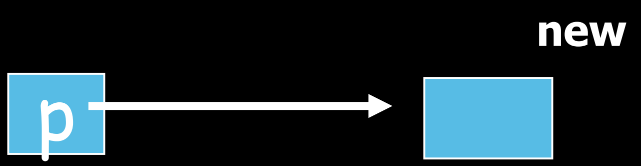
Example

Dynamic Memory Allocation

- Request for “unnamed” memory from the Operating System

```
int *p, n=10;

p = new int;
```




```
p = new int[100];
```


```
p = new int[n];
```

Memory Allocation Example

Want an array of unknown size ?

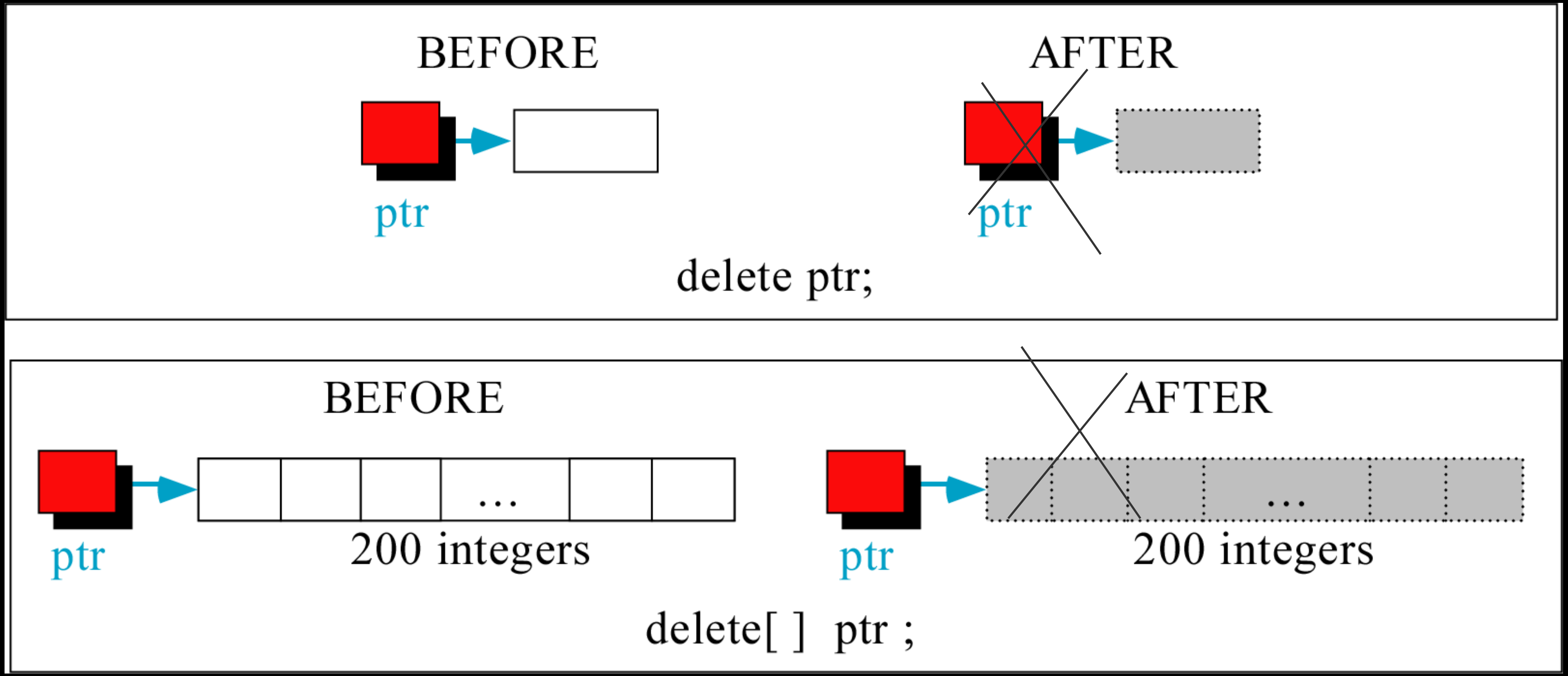
```
main()
{
     cout <<  “How many students? “;
     cin   >> n;

     int *grades = new int[n];

     for(int i=0; i < n; i++){
          int mark;
          cout << “Input Grade for Student” << (i+1)  << “ ? :”;
          cin >> mark;
          grades[i] = mark;
     }

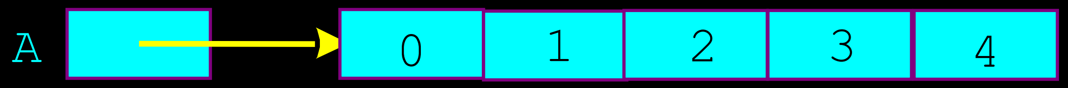
    . . .
    printMean( grades, n ); // call a function with dynamic array
    . . .
 }
```

Freeing (or deleting) Memory



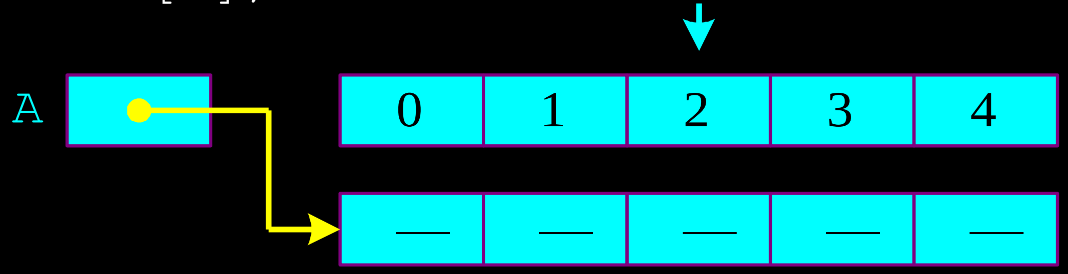
Caution 1: Memory Leak Problem

```
int *A = new int [5];
for(int i=0; i<5; i++)
   A[i] = i;
```



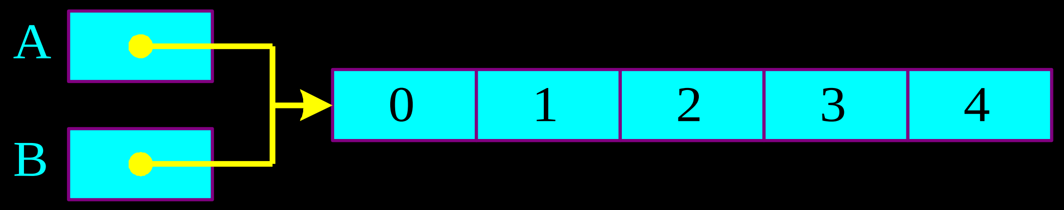

```
A = new int [5];
```

These locations cannot be accessed by program

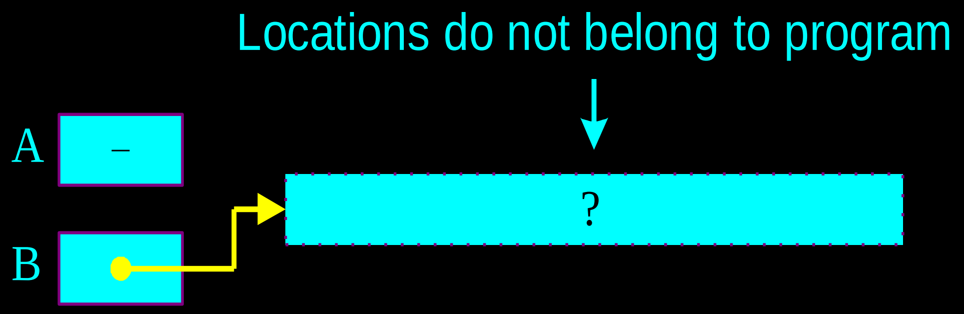


Caution 2; Dangling Pointer Problem

```
int *A = new int[5];
for(int i=0; i<5; i++)
   A[i] = i;
int *B = A;
```




```
delete [] A;
B[0] = 1; // illegal!
```



5. Structures

Structure

- A structure is a user-defined data type in C/C++.
- It is used to store together elements of different data types.
- A structure creates a data type that can be used to group items of possibly different types into a single type.

When we may use structures ?

- Suppose you need to store information about a student; like name, cgpa, and age.
  - You can create variables like name, cgpa, and age to store the data separately.
- However, you may need to store information about many students in the future.
  - It means variables for different individuals will be created.
  - For example, name1, cgpa1, age1 etc.
- To avoid this, it's better to create a struct AND array of struct.

How to create a structure?

- The **'struct'** keyword is used to create a structure.
- The **general syntax** to create a structure is:

```
struct structureName{
    member1;
    member2;
    .
    .
    .
    memberN;
} [zero or more structure variables];
```

Example struct declaration

```
struct Student
{
   char name[30];
    float cgpa;
    int age;
}S1;
```

In the above example, Student is a structure with three members. And an instance(variable) of the Student structure **S1**

Note;

- Memory is only allocated after a variable is added to the struct.

How to declare structure variables?

- A structure variable can be declared in either of the two ways
  1. With structure declaration or
     - Example:
       – look at the struct declaration on the previous slide, s1 variable is declared during struct declaration
  2. As a separate declaration like basic types.
- Example

```
int main(){
   Student s2;
   return 0;
}
```

How to initialize structure members?

- Prior to C++ 11, Structure members **cannot be** initialized with declaration.
- Structure members **can be** initialized using curly braces '{}'. For example, following is a valid initialization.

```
int main()
{
/*during assigning values to member variables, the order of declaration is followed. */
Student s1 = {"Abebe Kebede", 3.75, 20};
}
```

How to access structure elements?

- Structure members are accessed using dot (.) operator.
- Example

```
int main(){

/*during assigning values to member variables, the order of declaration is followed. */

student s1 = {"Abebe Kebede", 3.75, 20};

S1.cgpa=3.89;

cout<<"cgpa of "<<s1.name<<"is"<<s1.cpga;

}
```

Pointers to Structure

- It's possible to create a pointer that points to a structure.
- It is similar to how pointers pointing to native data types like int, float, double, etc.
- Example

```
int main(){
student s1 = {"Abebe Kebede", 3.75, 20};
Student *sp=&s1;
sp->cgpa=3.89;
cout<<"cgpa of "<<sp->name<<"is"<<sp->cpga;
}
```

- The '.' operator can also be used with struct pointer, but can look clumsy;
- Example:
  `cout<<(*sp).cgpa ;`
- Note:
  - () is required due to operator precedence

Another Example: structure pointers

```
#include <iostream>
#include <cstring>

using namespace std;

struct student
{
  string name;
  int roll_no;
};

int main(){

    struct student stud = {"Sam",1};
    struct student *ptr;
    ptr = &stud;

    cout << stud.name << stud.roll_no << endl;
    cout << ptr->name << ptr->roll_no << endl;
    return 0;
}
```

Exercise: What is the output ?

```
#include <iostream>
#include <cstring>

using namespace std;

int main(){

    struct student
    {
        int roll_no;
        string name;
        int phone_number;
    };

    struct student p1 = {1,"Brown",123443};
    struct student p2, p3;
    p2.roll_no = 2;
    p2.name = "Sam";
    p2.phone_number = 1234567822;
    p3.roll_no = 3;
    p3.name = "Addy";
    p3.phone_number = 1234567844;
```

```
    p3.phone_number = 1234567844;

    cout << "First Student" << endl;
    cout << "roll no : " << p1.roll_no << endl;
    cout << "name : " << p1.name << endl;
    cout << "phone no : " << p1.phone_number << endl;
    cout << "Second Student" << endl;
    cout << "roll no : " << p2.roll_no << endl;
    cout << "name : " << p2.name << endl;
    cout << "phone no : " << p2.phone_number << endl;
    cout << "Third Student" << endl;
    cout << "roll no : " << p3.roll_no << endl;
    cout << "name : " << p3.name << endl;
    cout << "phone no : " << p3.phone_number << endl;
    return 0;
}
```

Array of Structures

- On the previous example of structures, we stored the data of 3 students.
- Now suppose we need to store the data of 100 such students.
  - Declaring 100 separate variables of the structure is definitely not a good option.
  - For that, we need to create an **array of structures**.

Array of Structures

- ```
  struct student
  {
     int roll_no;
     string name;
     int phone_number;
  };
  int main()
  {
     student stud[100];

           ...


     return 0;
  }
  ```
  ```

Example: read and display details of 5 students

```
#include <iostream>
#include <cstring>

using namespace std;

struct student
{
  int roll_no;
  string name;
  int phone_number;
};

int main(){

    struct student stud[5];
    int i;

    for(i=0; i<5; i++){
        cout << "Student " << i + 1 << endl;
        cout << "Enter roll no" << endl;
        cin >> stud[i].roll_no;
        cout << "Enter name" << endl;
        cin >> stud[i].name;
        cout << "Enter phone number" << endl;
        cin >> stud[i].phone_number;
    }
```

```
    }

    for(i=0; i<5; i++){                    //printing values
        cout << "Student " << i + 1 << endl;
        cout << "Roll no : " << stud[i].roll_no << endl;
        cout << "Name : " << stud[i].name << endl;
        cout << "Phone no : " << stud[i].phone_number << endl;
    }
    return 0;
}
```

Nested structures

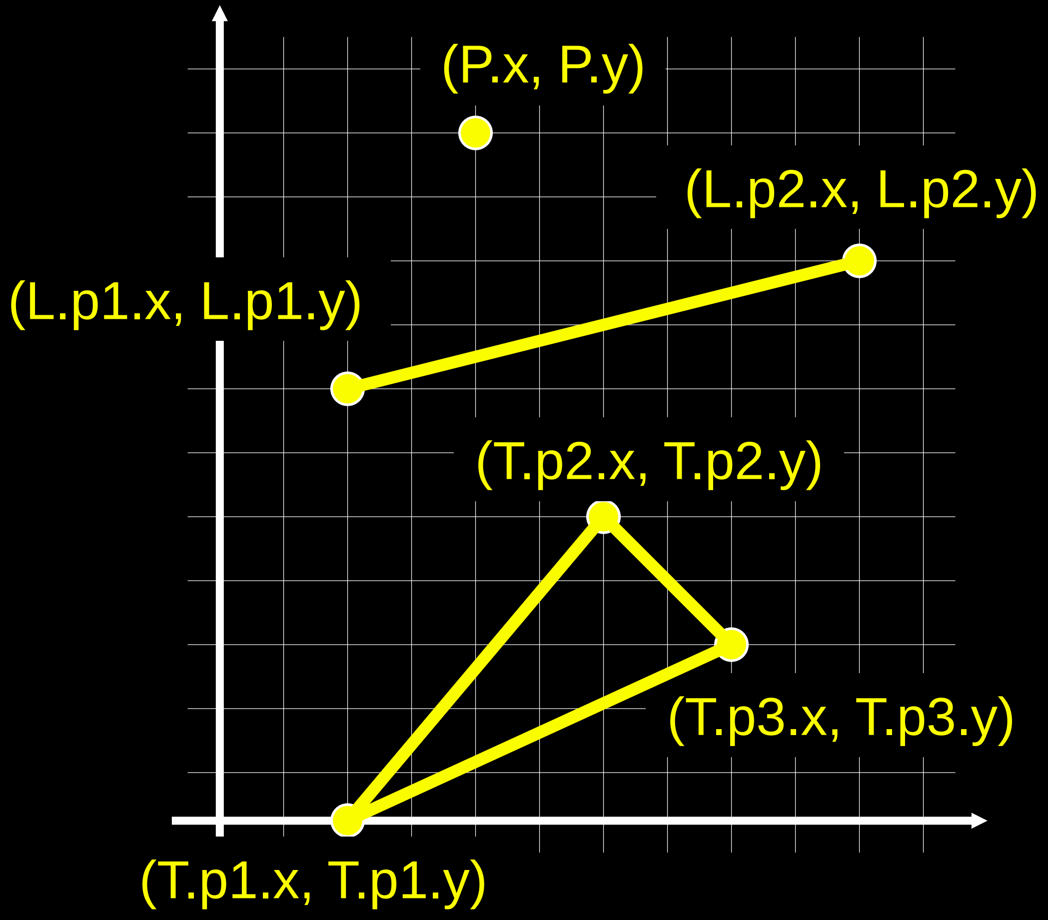
- We can nest structures inside structures.

- Examples:

```
struct point{
      double x, y;
};
point P;

struct line{
      point p1, p2;
};
line L;

struct triangle{
      point p1, p2, p3;
};
triangle T;
```

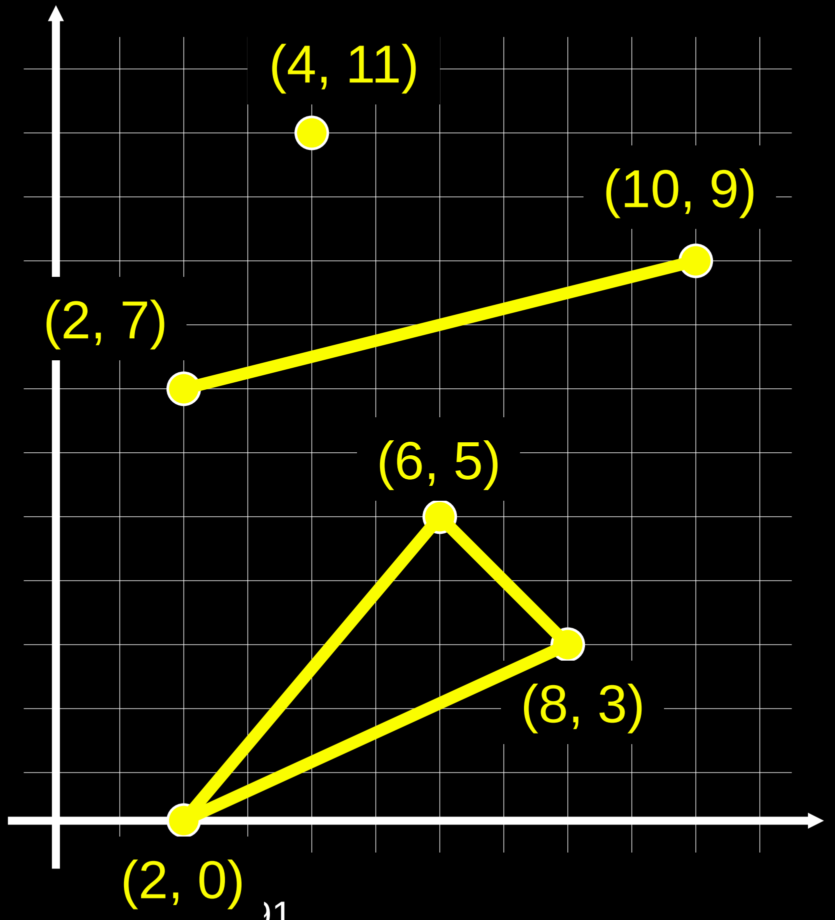


Nested structures

```
point P;
line L;
triangle T;

P.x = 4;

P.y = 11;
L.p1.x = 2;
L.p1.y = 7;
L.p2.x = 10;
L.p2.y = 9;
T.p1.x = 2;
T.p1.y = 0;
T.p2.x = 6;
T.p2.y = 5;
T.p3.x = 8;
T.p3.y = 3;
```



C++ Structure and Function

- Structures variables can be passed to a function and returned in a similar way as normal arguments.

Example:

```
struct_type func_name(struct_type){
.
.
.
return struct_type;
}
```

Passing and returning structure variables to/from functions

```
struct Person
{   char name[50];
   int age;
   float salary;
};
Person getData(Person); // Function declaration
void displayData(Person);
int main()
{
  Person p;
 // Function call with structure variable as an argument
   p = getData(p);
   displayData(p);
   return 0;
}
```

```
// Function defination
Person getData(Person p) {
    cout << "Enter Full name: ";
    cin.get(p.name, 50);
    cout << "Enter age: ";
    cin >> p.age;
    cout << "Enter salary: ";
    cin >> p.salary;
    return p;
}
void displayData(Person p)
{
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p.name << endl;
    cout <<"Age: " << p.age << endl;
    cout << "Salary: " << p.salary;
}
```

Next Time!
Ch2: Complexity Analysis