Lecture 5

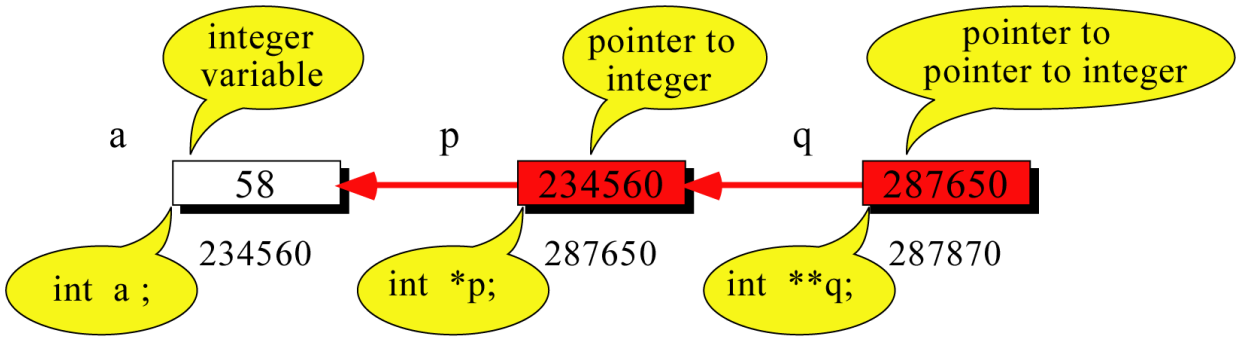
Dynamically allocating 2D Array + Passing arguments to function

DYNAMICALLY ALLOCATING 2D ARRAYS

First thing: Pointer to Pointer

- A pointer to a pointer **"Holds the address of another pointer."**
- It is declared using to asterisk, int **ptr

```
// 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

Pointer to Pointer

- A pointer to a pointer works just like a normal pointer — you can perform indirection through it to retrieve the value pointed to.
- And because that value is itself a pointer, you can perform indirection through it again to get to the underlying value.
- These indirections can be done consecutively, For example:

```
int value = 5;

int *ptr = &value;
std::cout << *ptr; // Indirection through pointer to int to get
int value

int **ptrptr = &ptr;
std::cout << **ptrptr; // first indirection to get pointer to int
, second indirection to get int value
```

The program prints:

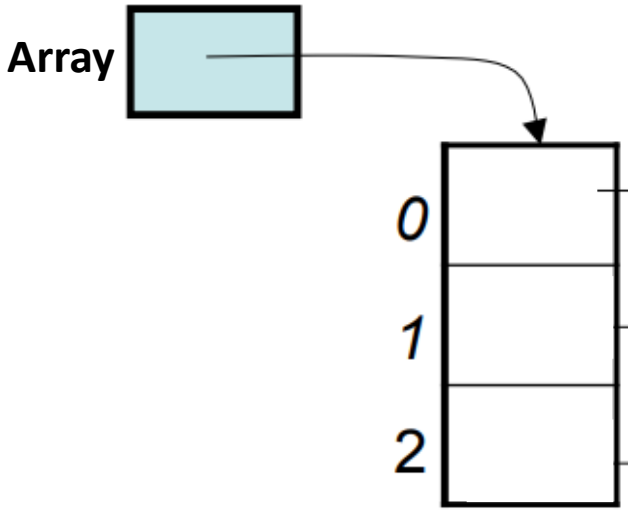
5 5

2nd thing: Array of Pointers

- Pointers to pointers have a few uses.
- The most common use is to dynamically allocate an array of pointers:

```
int **array = new int*[3]; // allocate an array of 10 int pointers
```

- This works just like a standard dynamically allocated array, except the array elements are of type "***pointer to integer***" instead of integer



How to dynamically allocate 2D Arrays

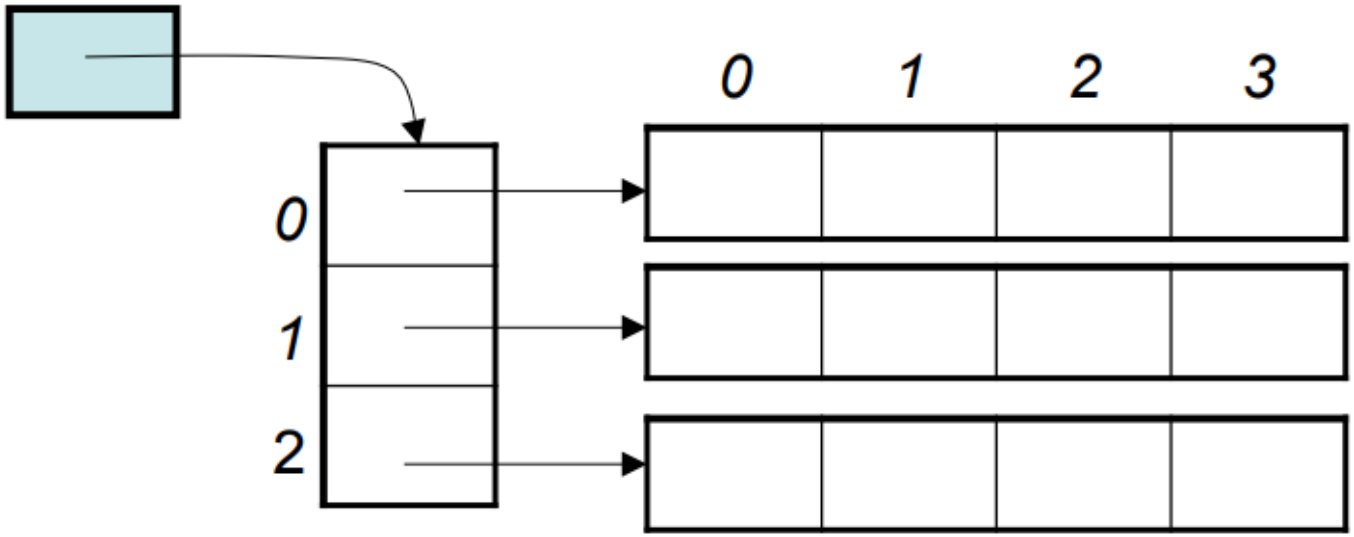
- Basic Idea:
  1. Allocate an array of pointers (first dimension), → just like we did in previous slide

     int **array = new int*[3];

  2. Make each pointer point to a 1D array of the appropriate size (2nd Dimension)

     array[count] = new int[4]; // these are our columns

- Now each row has 4 columns.

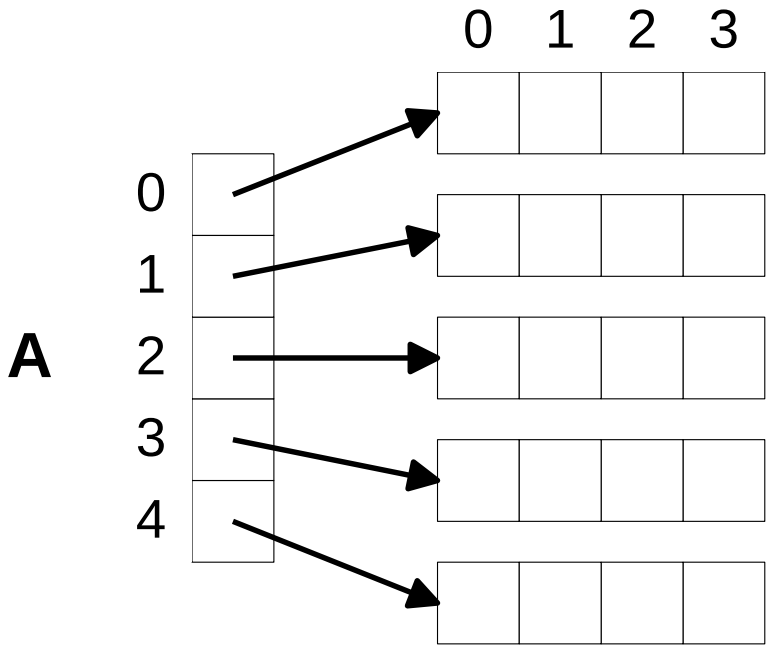


Assigning values to Dynamically Allocated 2D Arrays

- Can treat result as 2D array

- We can then access our array like usual:

array[3][2] = 3; // This is the same as (array[3])[2] = 3



Let's put it together

- Our dynamic two-dimensional array is a dynamic one-dimensional array of dynamic one-dimensional arrays!

```
int **array = new int*[10]; // allocate an array of 10 int pointers — these are our rows

for (int count = 0; count < 10; ++count)
    array[count] = new int[5]; // these are our columns
```

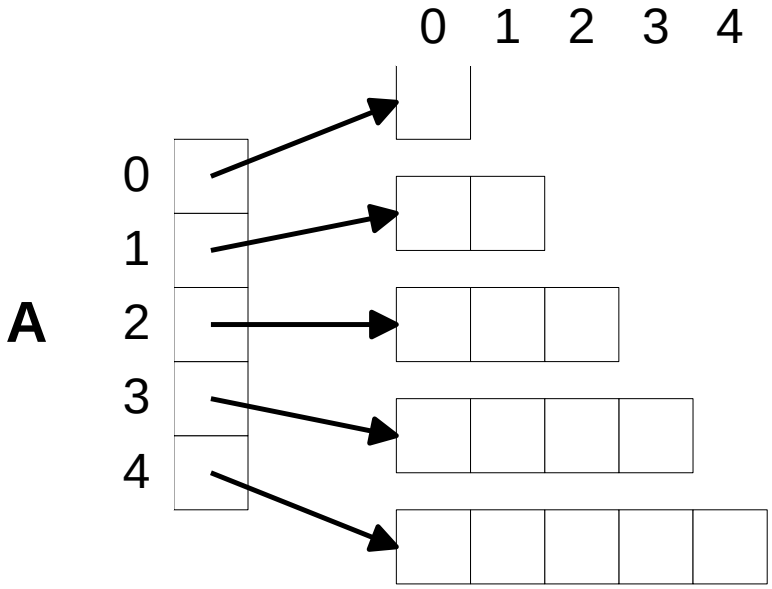
- We can assign values to 2d-array using array[3][2] = 3;

Non-Square 2D Arrays

- No need to allocate square 2D arrays:
- it's possible to make dynamically allocated two dimensional arrays that are **not rectangular**.
- For example, the following code can make a **triangle-shaped array**:

```
int **array = new int*[10]; // allocate an array of 10 int pointers-these are our rows
for (int count = 0; count < 10; ++count)
    array[count] = new int[count+1]; // these are our columns
```

- array[0] is an array of length 1,
- array[1] is an array of length 2,
- etc…



Memory Deallocation

- Deallocating a dynamically allocated 2d array using this method requires a loop as well, i.e,
  - Each row must be deleted individually
- Be careful to delete each row before deleting the array pointer.

Step 1:

```
for(int i=0; i<6; i++)
        delete [ ] array[i];
```

Step 2:

```
delete [ ] array;
```

- **Note** that we delete the array in the opposite order that we created it (elements first, then the array itself).

Introduction to Functions, Function Parameters and Arguments

Functions: What are they?

- We can write our own functions in C++
- These functions can be called from your main program or from other functions
- A C++ function consists of a grouping of statements to perform a certain task
- This means that all of the code necessary to get a task done doesn't have to be in your main program
- You can begin execution of a function by calling the function

Functions: What are they?

- A function has a name assigned to it and contains a sequence of statements that you want executed every time you invoke the function from your main program!
- Data is passed from one function to another by using arguments (in parens after the function name).
- When no arguments are used, the function names are followed by: "()".

Functions: Defining Them...

- The syntax of a function is very much like that of a main program.
- We start with a function header:

```
data_type function_name()
{
    <variable definitions>
    <executable statements>
}
```

Functions: Defining Them...

- A function must always be declared before it can be used
- This means that we must put a one-line function declaration at the beginning of our programs which allow all other functions and the main program to access it.
- This is called a **function prototype** (or **function declaration**)
- The function itself can be defined anywhere within the program.

Functions: Using Them…

- When you want to use a function, it needs to be CALLED or INVOKED from your main program or from another function.
- If you never call a function, it will never be used.
- To call a function we must use the function call operator ()

```
some_variable = pow (x, 3);
```

Functions: Calling pow...

- When we call a function, we are temporarily suspending execution of our main program (or calling routine) and executing the function.
- `pow` takes two values as arguments (`x` and `3`), called **actual arguments** and returns to the calling routine the result (a floating point value)

Introduction to function parameters and arguments

- **Parameters vs Arguments**
- In common usage, the terms parameter and argument are often interchanged.
- A function parameter (sometimes called a formal parameter) is a variable declared in the function declaration:

  void square(int x); // declaration (function prototype) -- x is a parameter

- An argument (sometimes called an actual parameter) is the value that is passed to the function by the caller:

  square(6); // 6 is the argument passed to parameter x

Ways to pass arguments to the function…

1. Pass by Value
2. Pass by Reference
3. Pass by Address

Passing arguments by value

- When the function call is executed,
- the actual arguments are conceptually copied into a storage area local to the called function.
- If you then alter the value of a formal argument, only the local copy of the argument is altered.
- The actual argument never gets changed in the calling routine.

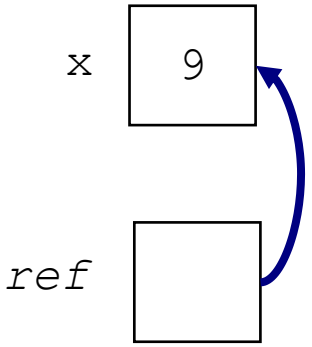
Let's write a function to sum two numbers:

```
int sumup(int first, int second);  //function prototype
void main() {
 int total, number, count;
 total = 0;
 for (count = 1; count <= 5; count++) {
     cout <<" Enter a number to add: ";
     cin >>number;
     total = sumup(total, number); //function call
 }
 cout <<" The result is: " <<total <<endl;
}
int sumup(int first, int second) { //definition
 return  first + second;
}
```

Reference Variables

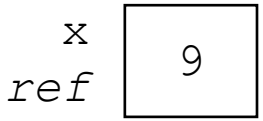
A reference is an additional name to an existing memory location

Pointer:




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

Reference:




```
int x = 9;
int &ref = x;
```

Note how the pointer necessitates an extra variable, whereas the reference didn't.

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
```

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

Pass arguments by Reference

- While pass by value is suitable in many cases, it has a couple of limitations.
- First, when passing a large struct or class to a function, pass by value will make a copy of the argument into the function parameter.
- when passing arguments by value, the only way to return a value back to the caller is via the function's return value.

Example of call by reference:

```
void convert (float inches, float & mils);
int main() {
     float in; //local variable to hold # inches
     float mm; //local variable for the result
     cout <<"Enter the number of inches: ";
     cin >>in;
     convert (in, mm);   //function call
     cout <<in <<" inches converts to " <<mm <<"mm";
     return 0;
  }
void convert (float inches, float & mils) {
     mils = 25.4 * inches;
  }
```

Example of call by reference:

```
void swap (int & a, int & b);
int main() {
 int i=7, j = -3;
 cout <<"i and j start off being equal to :" <<i
      <<" & " <<j <<'\n';
 swap(i,j);
 cout <<"i and j end up being equal to    :" <<i
      <<" & " <<j <<'\n';
 return 0;
}
void swap(int &c,int&d) {
 int temp = d;
 d = c;
 c = temp;
}
```

What kind of args to use?

- Use a call by reference if:

  1) The function is supposed to provide information to some other part of the program. Like returning a result and returning it to the main.

  2) They are OUT or both IN and OUT arguments.

  3) In reality, use them WHENEVER you don't want a duplicate copy of the arg...

What kind of args to use?

- Use a call by value:

  1) The argument is only to give information to the function - not get it back

  2) They are considered to only be IN parameters. And can't get information back OUT!

  3) You want to use an expression or a constant in function call.

  4) In reality, use them only if you need a complete and duplicate copy of the data

Passing arguments by address

- Passing an argument by address involves passing the address of the argument variable rather than the argument variable itself.
- Since the argument is an address, the function parameter must be a pointer.
- The function can then dereference the pointer to access or change the value being pointed to.

Example of call by address

```
#include <iostream>
 void foo(int *ptr)
{
   *ptr = 6;
}
 int main()
{
   int value =  5 ;
    cout << "value = " << value << '\n';
   change(&value); //Function call, pass by address
   cout << "value = " << value << '\n';
   return 0;
}
```

```
Output is:

value = 5

value = 6
```

When to use call by address?

- **Advantages:**
- Pass by address allows a function to change the value of the argument, which is sometimes useful.
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- We can return multiple values from a function via out parameters.
- **Disadvantages:**
- Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by value.
- All values must be checked to see whether they are null. Trying to dereference a null value will result in a crash. It is easy to forget to do this.

Finally, pass by address and pass by reference have almost identical advantages and disadvantages. Because pass by reference is generally safer than pass by address, pass by reference should be preferred in most cases.