Object-oriented programming (with C++)

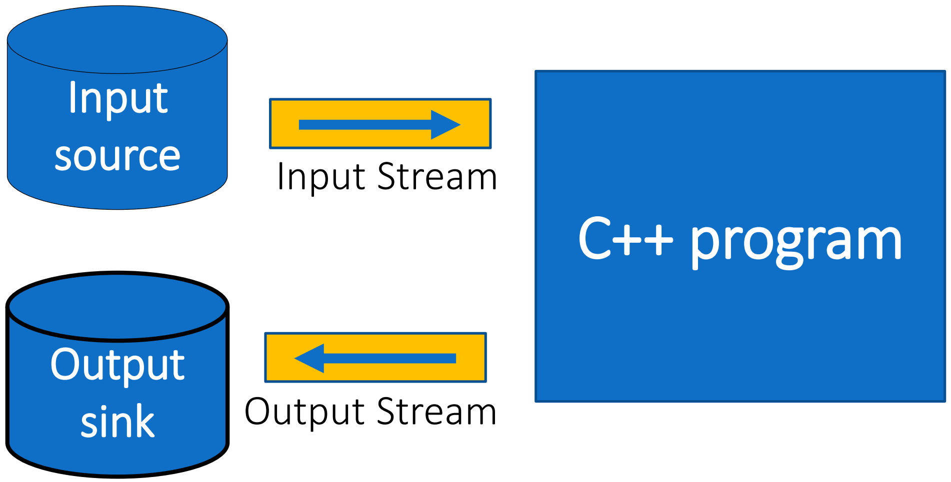
Lecture #2: Overview of basic structures in C++

Non Object-Oriented Extensions to c

- Major improvements over C
  - Stream I/O
  - Declarations
  - Parameter passing by reference
  - Default argument values

Stream I/O in C++

- Input and output in C++ can be handled by streams.
- A stream is a sequence of bytes that may be either *input* to a program or *output* from a program



- **cin**, **cout**, and **cerr** are three standard devices for input (keyboard), output and error output (tied to the screen)
- **iostream** should be included (using namespace std)

I/O Stream in C++

- Buffered (***cin, cout***)
- Unbuffered (***cerr***)
- Buffered characters should be flushed
- Unbuffered characters can be seen immediately

I/O Stream in C++

- The directive **`#include <iostream.h>`** declares 2 streams: **`cin`** and **`cout`**
- **`cin`** is associated with standard input.
  - Extraction: **`operator>>`**
- **`cout`** is associated with standard output
  - Insertion: **`operator<<`**

Simple example (cout)

```
// A Simple Hello-World Program
#include <iostream>
main() {
    /* the output statement */
    std::cout << "Hello World! \n";
}//ex1-hello.cpp
```

Simple example (cout)

```
#include <iostream>
using namespace std;
main() {
    int a = 15, b = 10;
    cout << "a = " << a << ", b = " << b << "\n";
    float f = 3.14;
    cout << "f=" << f;
    cout << 5 << ", " << 26.5 << ", " << (a+b) << '\n';
    char ch ='A'; int n = 65;
    cout << ch << ", " << int(ch) << ", " << char(n) << endl;
}//ex2-cout.cpp
```

Simple example (cin)

```
#include <iostream>
using namespace std;
main() {
	int a, b;
	float f; char ch;
	cout << "Enter two integers, one float, and a char: ";
	cin >> a >> b >> f >> ch;
	cout << "a = " << a << ", b = " << b << ", f = " << f << ", ch = " << ch << endl;
	char name[80];
	ch = '\0';
	int i = 0;
	cout << "Enter your name (with '#' at the end): \n";
	while (1) {
	  cin >> ch; // ch = cin.get()
	  if (ch == '#') break;
	  name[i++] = ch;
	}
	name[i]= '\0';
	cout << name << endl;
}//ex3-cin.cpp
```

Simple example

```
#include <iostream>
#include <fstream>
int fileSum();
int fileSum()
{
        std::ifstream infile("Ex2file");
        int value, n;
        int sum=0;
        //Read the number of integers
        infile>>n;
        //Read members
        int i=0;
        while (i<n)
        {
                infile>>value;
                sum=sum+value;
                i++;
        }
        return sum;
        infile.close();
}
```

Why use I/O streams

- Streams are sub-classable; istreams, ostreams,… are real classes and hence sub-classable. We can define types that look and act like streams, yet operate on other objects. Examples:
  - Stream that listens to external port
  - Stream that writes to a memory area.

Why use I/O streams

- Streams may be faster: **`printf`** interprets the language of '%' specs, and chooses (at runtime) the proper low-level routine. C++ picks these routines statically based on the actual types of the arguments.
- Streams are extensible; I/O mechanism is extensible to new user-defined data types.

Declarations in C++

- In C++, declarations can be placed anywhere (except in the condition of a **`while`**, **`do/while`**, **`for`** or **`if`** structure.)
- An example

```
cout << "Enter two integers:";
int a, b;
cin >> a >> b;
cout << "The sum of" << a << " and " << b
    << " is " << x + y << endl;
```

Parameter passing by reference

- Parameters: variables in the method definition (the member + the type)
  - int f(int x) {…}

- Arguments: the actual value of this variable that gets passed to function. The member in the calling
  - f(m)

Parameter passing by reference

- In C, all function calls are call by value.
  - Call be reference is simulated using pointers.

- *Reference parameters* allows function arguments to be changed without using return or pointers.

Call by value

```
#include <iostream>
using namespace std;
int incrementByValue(int);
main()
{
        int x = 5;
        cout << "x = " << x << " before incrementByValue\n"
                << "Value returned by sqrByVal: "
                << incrementByValue(x)
                 << "\nx = " << x << " after incrementByValue\n\n";
}
int incrementByValue(int a)
{
        return a += 10;
        // caller's argument not modified
}
```

Call by Reference

```
#include <iostream>
using namespace std;
int incrementByRef(int &);
main()
{
        int z = 9;
         cout << "z = " << z << " before incrementByRef\n";
                incrementByRef(z);
                cout << "z = " << z << " after incrementByRef\n";
}
int incrementByRef(int &a)
{
        return a += 10;
        // caller's argument modified
}
```

Constant Reference Parameter

```
#include <iostream>
using namespace std;
void incrementByRef(const int &);
main()
{
        int z = 9;
         cout << "z = " << z << " before incrementByRef\n";
                incrementByRef(z);
                cout << "z = " << z << " after incrementByRef\n";
}
int incrementByRef(const int &a)
{
        return a += 10;
        // caller's argument modified
}
```

Default Arguments

- Parameters can be assigned default values
- Parameters assume their default values when no actual parameters are specified for them in a function call

Example

```
#include <iostream>
using namespace std;
int sum (int lower, int upper=10, int inc=1) {
  int sum = 0;
  for (int k = lower; k <= upper; k+= inc)
        sum += k;
  return sum;
}
main()
{
        cout << "Sum of integers from 1 to 10 is " << sum (1);
}//ex6-defaultarg.cpp
```

Pointers and Dynamic Memory Allocation

- Overviews
- Pointer Conversions
- Allocating Memory
- Arrays and Dynamic Allocation

Typical ways to declare a pointers

- Pointers
  - A **pointer** is a variable whose value is the address of another variable.

- Typical ways to declare a pointers:
  - char *p=message;
  - char * const p=message;
  - const char *p=message;
  - const char * const p=message;

Pointers to Constants

- **Pointers to Constants**: The object cannot be modified when this pointer is used for access

```
int x = 0;
const int * p = &x;
*p = 30; // Error!
x = 10; // OK!
```

Constant Pointer

```
char * const aStr=“John Smith”;
char msg[10]=“John Smith”;


char * const sp=msg;
sp++; //Error;
strcpy(sp, “Nam”); //OK
```

Pointer and function

- Cannot pass a pointer to a constant to a function with a parameter that is a pointer to a non-constant

```
char * aFunction(char *, const char *);
main()
{
    const char *a;
    const char *b;
     aFunction(a,b); //Error!
}
```

Pointer and function

- Pointer arguments:
  - Pass-by-reference.

Examples

Pointer conversions

- Pointers to Array Elements
- Void Pointers
- References to Pointers

Pointers to Array Elements

- Pointer is related to a type or a class
- The pointer pointing to an array can be incremented or decremented (pointer arithmetic)

```
float flist[] = { 10.3, 13.2, 4, 9.6 };
float * fp = flist;
fp++;
cout << *fp;
```

Pointers to Array

```
float flist[] = {10.3, 13.2, 4, 9.6};
float * fp = flist;
int * ip = (int *) fp;
ip++;
cout << *ip;
```

Void Pointers

- To obtain flexibility of types

```
void * memcpy (void *dest, const void * src, size_t nbytes);

const unsigned ArraySize = 500;
long arrayOne[ArraySize];
long arrayTwo[ArraySize];
memcpy (arrayOne, arrayTwo, ArraySize * sizeof(long));
```

Void Pointers

- Casting required

```
int *p;
void *v = p;
p = (int *) v; //cast needed
```

Allocating Memory

- Static and Dynamic Allocation
  - Allocating memory for objects is handled by the compiler– **Static Allocation**
  - Allocating memory for objects at run time and you can control the exact size and the lifetime of these memory locations – **Dynamic Allocation**
  - **Automatic memory allocation:** occurs for (non-static) variables defined inside functions.

Static and Dynamic Allocation

- De-allocating memory refers to releasing a block of memory whose address is in a pointer variable – deleting a pointer.

- Fragmentation: is the condition where gaps occur between allocated objects

The *new* Operator

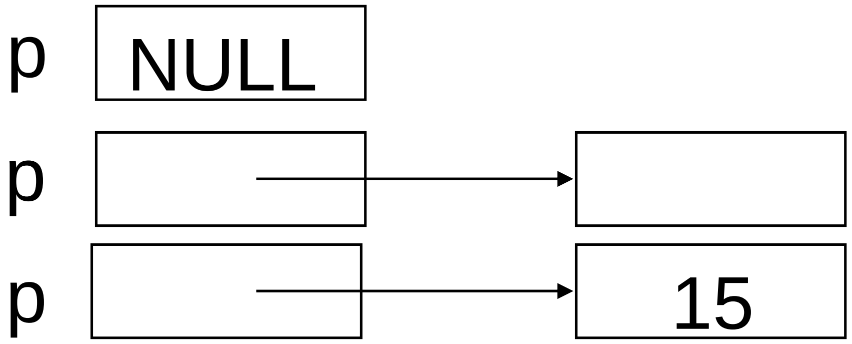
- The ***new*** operator is used to allocate memory dynamically

int *p;

p = ***new*** int;

*p = 15;

int * array = new int[50];



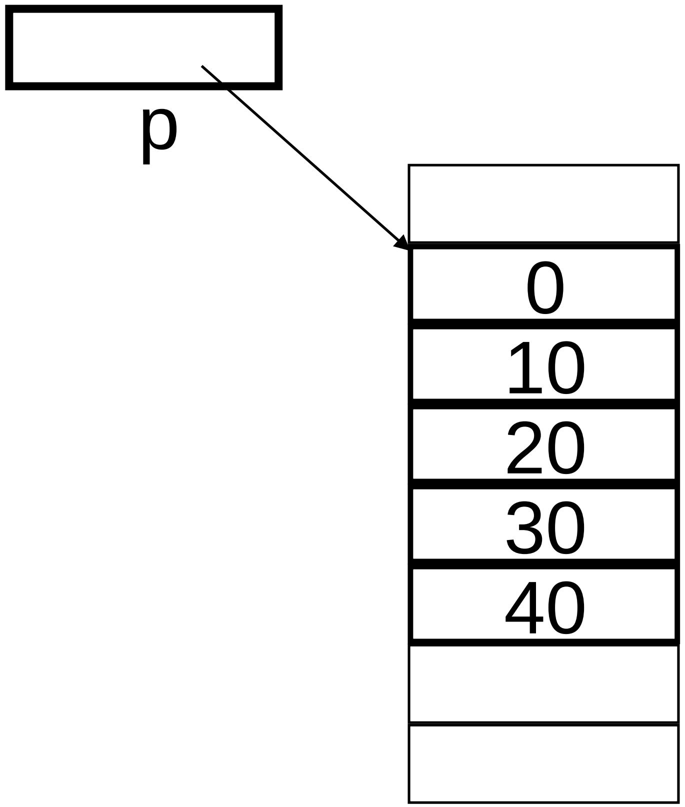
The *delete* Operator

- The ***delete*** operator is used to deallocate memory space (created dynamically)

```
delete  p;
delete [] array;
```

One-dimensional arrays

```
int *p = new  int[5] ;
for (int j=0; j < 5; ++j)
   *(p + j) = 10 * j;
for (int j=0 ; j < 5; j++ )
   cout  <<  p[ j ];
delete [] p;
//ex7_new1DArray.cpp
```

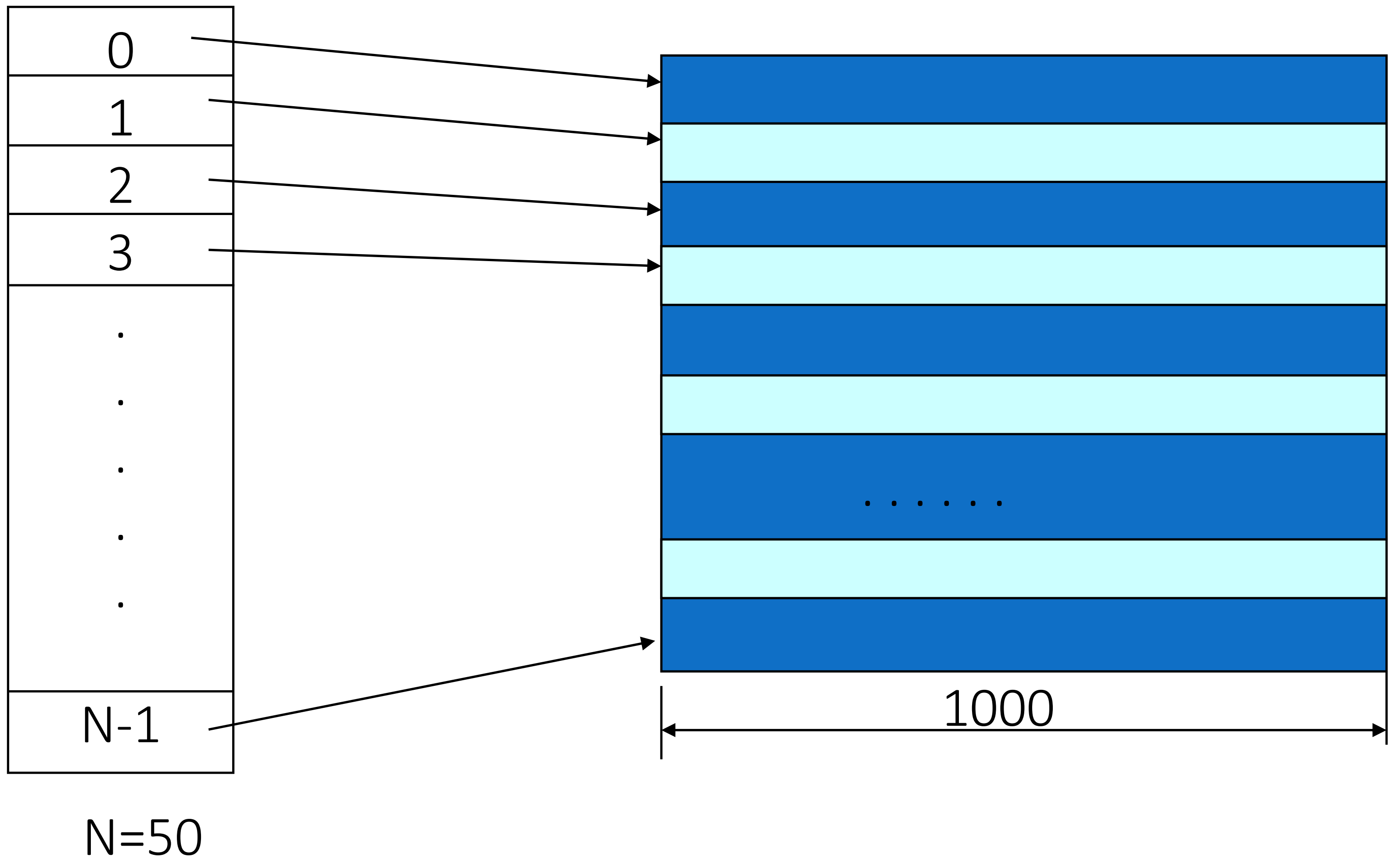


Two-Dimensional Arrays

- Array of Pointers

```
const unsigned NumRows = 50;
const unsigned RowSize = 1000;
int * samples[NumRows];
for (unsigned i = 0; I<NumRows; I++)
    samples[i]= new int[RowSize];//See the graph
```

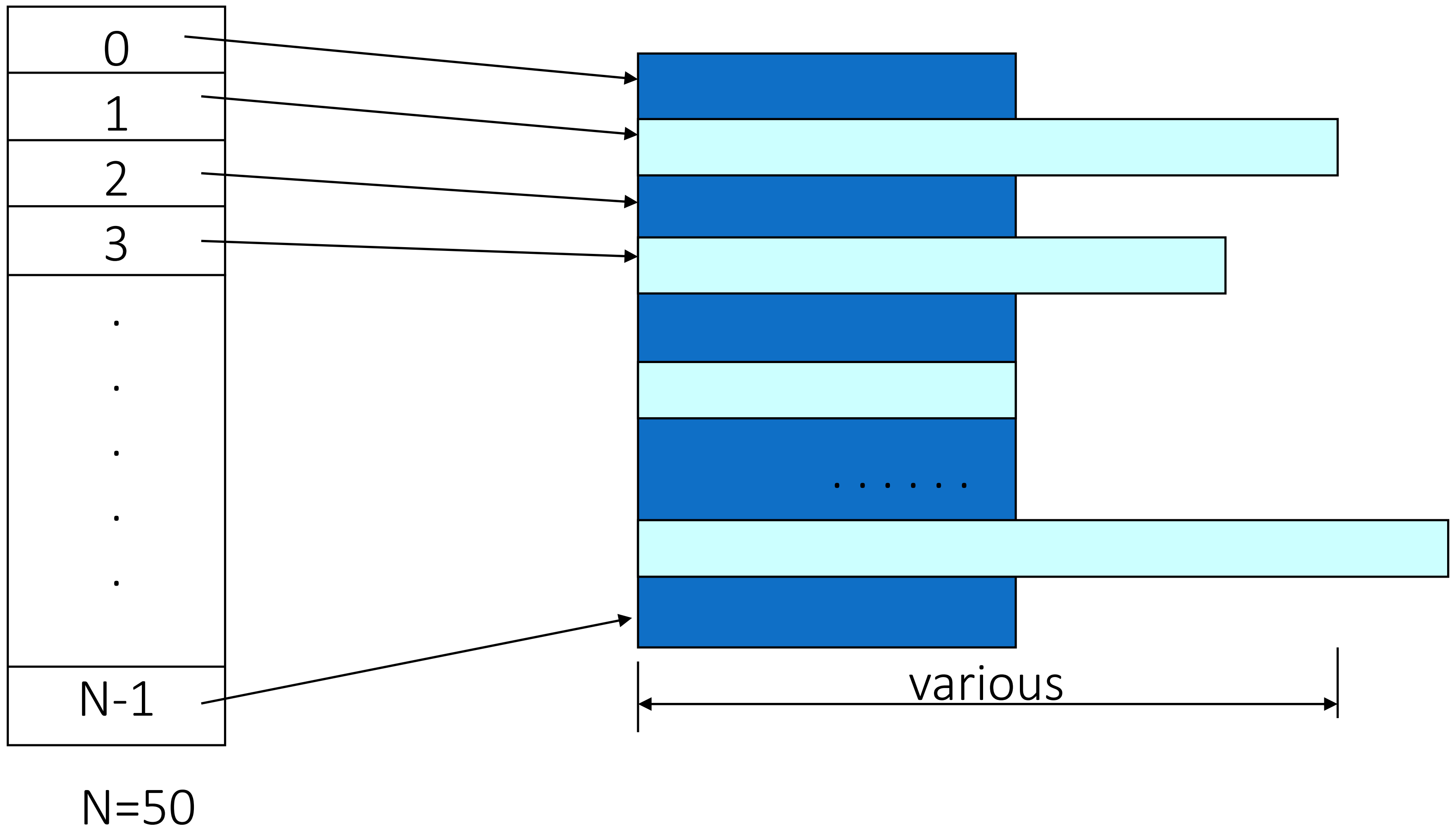
Array of Pointers



Accessing

```
for (unsigned i = 0; i<NumRows; i++)
   for (unsigned j = 0; j<RowSize; j++)
      samples[i][j]=0;
```

Ragged Array



Ragged Array

- Applications
  - To store an array of strings (e.g. our names)
  - Efficient storage for graph

Summary