EEL 4837
Programming for Electrical Engineers II

Ivan Ruchkin
Assistant Professor
Department of Electrical and Computer Engineering
University of Florida at Gainesville
iruchkin@ece.ufl.edu
http://ivan.ece.ufl.edu

C/C++ Pointers

Readings:

- Deitel chapters 6.13, 8
- Stroustrup chapter 7

What are Pointers?

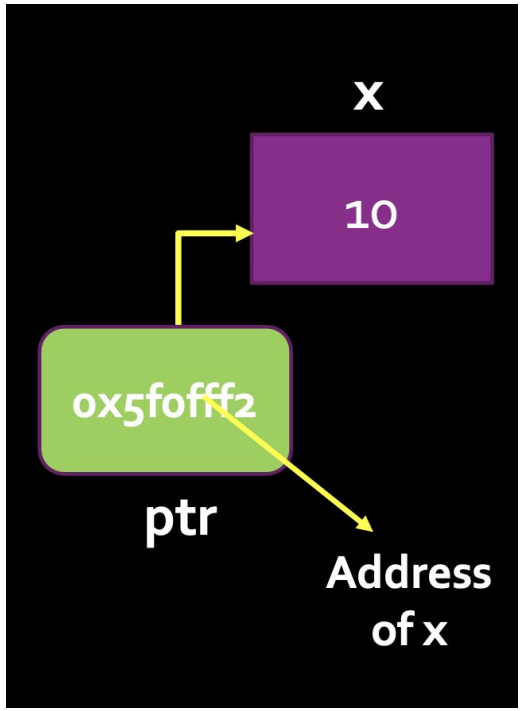
- A **pointer** is a variable that holds the memory address of another variable (of a specified type)
- This memory address is the location of another variable where it has been stored in the memory.

  Syntax: `Datatype* variableName;` (`Datatype *variableName;` also works)

  ```
  int* x; float* y; char* z;
  ```

Address of operator(&): is a unary operator that returns the memory address of its operand. Here the operand is a *normal variable*.

Dereferencing operator(*): is a unary operator that returns the value stored at the address pointed to by the pointer. Here the operand is a *pointer*.




```
int x = 10;
int* ptr = &x;

cout << ptr; // returns the value at ptr, i.e., 0xf50fff2
cout << *ptr; // returns the value at location 0xf50fff2, i.e, 10
```

Pointer Arithmetic

- Pointers support two arithmetic operations: **addition** and **subtraction**
- *When we add 1 to a pointer:* we are actually adding the size of data type in bytes, the pointer is pointing at
- Example:

```
int* x;
x++;
```

If current address of x is 1000, then x++ statement will increase x by 4 (size of int data type) and makes it 1004, not 1001.

Pointer Arithmetic++

***ptr++**	***(ptr++)**	**Increments pointer, and dereferences unincremented address i.e. This command first gives the value stored at the address contained in ptr and then it increments the address stored in ptr.**
***++ptr**	*(++ptr)	Increment pointer, and dereference incremented address i.e. This command increments the address stored in ptr and then displays the value stored at the incremented address.
++*ptr	++(*ptr)	Dereference pointer, and increment the value it points to i.e. This command first gives the value stored at the address contained in ptr and then it increments the value by 1.
(*ptr)++		Dereference pointer, and post-increment the value it points to i.e. This command first gives the value stored at the address contained in ptr and then it post increments the value by 1.

Constant Pointers

- C++ allows us to declare a pointer to be **constant**, whose value (the address) cannot be changed

```
int r = 5;
const int * rptr= &r;

rptr++;   // should give error in most C++ compilers
rptr = 3; // should give error in most C++ compilers
cout << *rptr  << endl; // OK
(*rptr)++; // OK

int* nptr = rptr;

nptr = 3; // perfectly ok
nptr++;   // perfectly ok
```

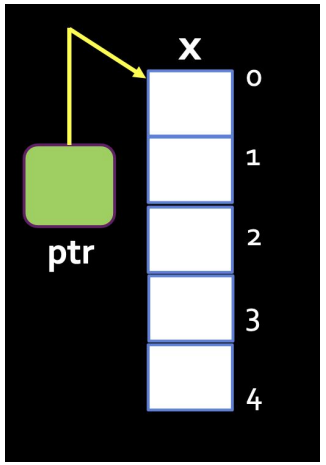
Pointers and Arrays

- C++ treats the name of an array as **constant pointer** which contains address of first location of array (also called *base address*)

```
int x[10];
```

- Here x is a constant pointer which contains the base address of the array x.

```
int x[5];
int* ptr = x; // ptr contains base address of x
int* ptr= &x[0]; // exactly the same as above

x++;    // error
ptr++; // pointer is advanced by size of int
       // so it points to the next array element
```



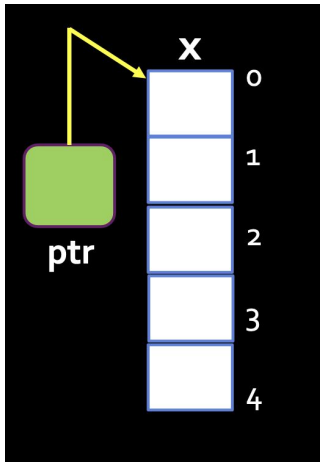
Pointers and Arrays

- C++ treats the name of an array as **constant pointer** which contains address of first location of array (also called *base address*)

```
int x[5];
```

- Here x is a constant pointer which contains the base address of the array x.

```
int x[5];
int* ptr = x; // ptr contains base address of x
int* ptr= &x[0]; // exactly the same as above

// Here are three ways to print array contents
for(int i=0;i<5;i++) cout << *(ptr+i);
for(int i=0;i<5;i++) cout << *ptr++;
for(int i=0;i<5;i++) cout << *(x+i);
```



Pointers and Strings

- A **C-style string** is just an array of characters ending in **'\0'**;
- Pointers work the same way as with other arrays

```
void main() {
   char str[] = "computer";
   char* cp=str;
   cout << str; // using variable name
   cout << cp;  // using pointer variable
}
```

The statement `cout<<cp;` will print the string `computer`, since if you give `cout` a pointer to a character it prints everything from that character to the first null character ('\0') that follows it.

Arrays of Pointers

- Like array of anything else, we can have **array of pointers**
- Commonly we use them to store strings

```
char* vehicle[] = {"CAR","VAN","CYCLE","TRUCK","BUS"};

for(int i=0; i<5; i++)
  cout << vehicle[i] << endl;
```

- `vehicle` is an array of pointers to character
- `vehicle[i]` is a pointer to character (which means it is a string)

Pass-By-Value

```
int test (int x) {
  x++;
  return 0;
}

int main () {
   int x = 100;
   cout << x << endl; // prints 100
   test(x);
   cout << x << endl;
   // still prints 100
}
```

By default, parameters are **pass-by-value** in C / C++

- The value of the actual argument is **copied** into the the formal parameter of the function during invocation.
- **Changes made to the parameter inside the function have no effect on the actual argument used to call the function**

But what will happen if the formal parameter is a pointer?

Pass-By-Pointer

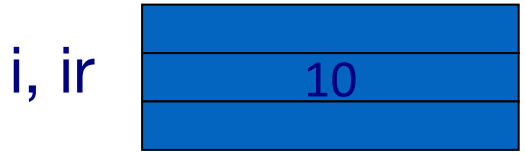
```
void swap(int *m, int *n)  {
 int temp;
 temp = *m;
 *m = *n;
 *n = temp;
}

void main() {
  int a = 5, b = 6;
  cout << "\n a :" << a << " and b: " << b;

  swap(&a, &b); // passing addresses of a and b
  cout << "\n a :" << a << " and b: " << b;
  // prints 6 5
}
```

- The **addresses** of actual arguments are copied into formal arguments of the called function
- Formal arguments are **pointers** and contain the address where the actual variables (a and b) are stored
- Therefore, in the called function whenever we change **the value at the location** pointed to by the argument pointer, the **stored values** will get changed and will be reflected after the function exits
- In the end, this is still **pass-by-value** but for pointer values

References (not to be confused!)

A **reference** is an ***alias*** for another variable:

```
int i = 10;
int &ir = i;    // reference (alias)
ir = ir + 1;    // increment i
ir = ir + 1;
```



Once initialized, references cannot be changed

Pass-By-Reference

- References are especially useful in ***function calls*** to avoid the overhead of passing arguments by value, without the clutter of explicit pointer dereferencing (e.g., `y = *ptr`)

```
void refInc(int &n){
    n = n+1; // increment the variable n refers to
}
```

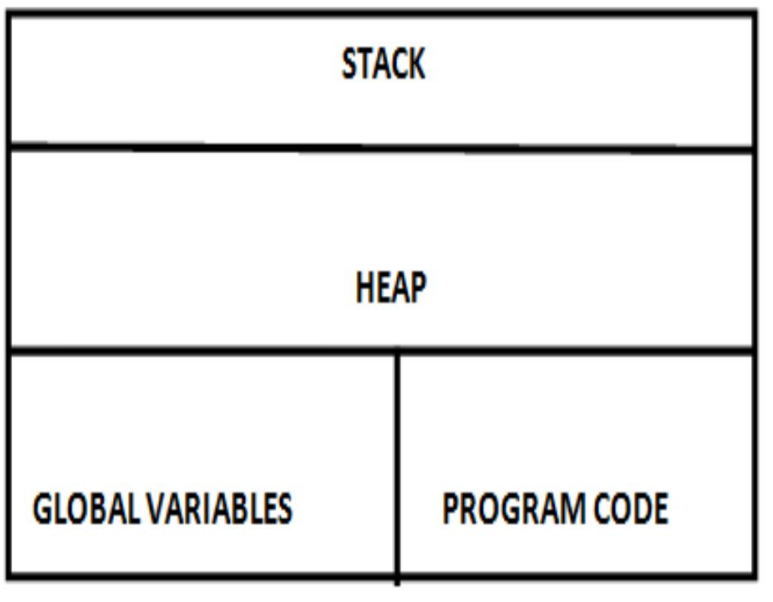
```
// swap by reference
void swap (int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
    return;
}
```

```
void main() {
    int a = 5;
    ont b = 10;
    refInc(a);
    cout << a << " " << b << endl;// 6 10
    swap(a, b);
    cout << a << " " << b << endl;// 10 6
}
```

C++ Memory Map

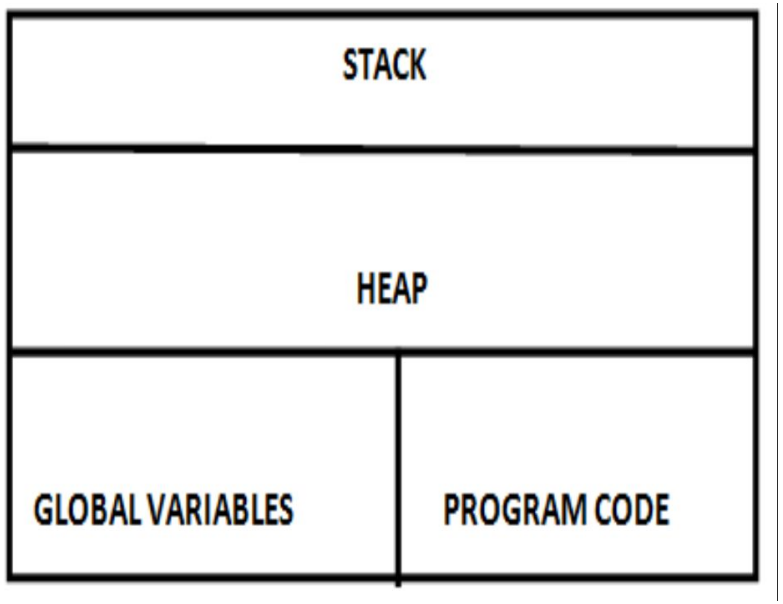
A **C++ program's memory** is divided into 4 parts:

- Program Code: it holds the compiled code of the program ready for execution
- Global Variables: they remain in the memory as long as program continues (includes a *program counter*)
- Stack: holds return addresses of function calls, arguments passed to the functions, local variables for functions (aka the *static memory*)
- Heap: a region of free memory from which chunks of memory are allocated



new and delete

- C++ dynamically allocates memory from the “free store”/“heap”/“pool”, i.e., the unallocated heap memory provided to the program by the operating system
- There are two unary operators new and delete that perform the task of allocating and deallocating memory during run time




```
int* i = new int; // allocates memory of size equal to the size of the operand (int)

char* ptr = new char ('A'); // can allocate and initialize at the same time

delete i; // deallocates the memory pointed to by the pointer variable i
```

Static vs Dynamic Memory Allocation

STATIC MEMORY ALLOCATION	DYNAMIC MEMORY ALLOCATION
The amount of memory to be allocated is known before hand.	The amount of memory to be allocated is not known before hand. It is allocated depending upon the requirements.
Memory allocation is done during compilation.	Memory allocation is done during run time.
For eg. int i; **This command will allocate two bytes of memory and name it 'I'.**	Dynamic memory is allocated using the new operator. For eg. int*k=new int; In the above command new will allocate two bytes of memory and return the beginning address of it which is stored in the pointer variable k.
The memory is deallocated automatically as soon as the variable goes out of scope.	To deallocate this type of memory delete operator is used. For eg. delete k;

Pointers to Structs

```
struct student {
    int rollno;
    char name[20];
};

void main() {
    student s1;
    cin >> s1.rollno;
    cin >> s1.name;

    student* stu;
    stu=&s1;     // now stu points to s1 i.e.
                 // the address of s1 is stored in stu

    cout << stu->rollno; // same as (*stu).rollno
}
```

Pointers to Objects

```
class student {
 int rollno;
 char name[20];
 public:
   // input student data
   void indata() {
     cin >> s1.rollno;
     cin >> s1.name;
   }
   // output student data
   void showdata() {
     cout<<stu.rollno<<" "<< stu.name;
   }
}
```

```
void main() {
   student s1, *stu;
   s1.indata();
   stu = &s1; //now stu points to s1
   s1.outdata();
   stu->outdata(); // same as above
}
```

Self-Referential Structures

A **self-referential structure** is a structure which contains an element that points/refers to the structure itself.

```
struct students {
   int rollno;
   char name[20];
   students* nextStudent; //pointer to structure itself
};
```

Self-referential structures are used for creating linked lists, trees, etc.

Dynamic Structures

```
struct student {
  int rollno;
   char name[20];
};

void main() {
  student* stu;
  stu = new student; // creating a new struct for student dynamically
  stu->rollno = 1;
  strcpy(stu->name, "Adam");
  cout << stu->roll << " " << stu->name;
  delete stu;  // deallocate the memory after we're done with students

  stu = NULL;  // can optionally assign NULL to remember the deletion
}
```

Dynamic Arrays

```
int * array = new int[10];  // create an array or size 10 dynamically
                            // array[0] refers to the first element
                            // array[1] refers to the second element

char* name = new char[20];   // create a string dynamically

int *arr, r=30, c=40;
arr = new int[r * c]; // created a "two-dimensional" array

// Really a one-dimensional array, but we can mimic operations of a
// two-dimensional array
for (int i = 0; i < r; i++) {
  for (int j = 0; j < c; j++) {
    cin >> arr[i+c*j]
  }
}

delete [] arr; // use delete[] for every new …[…]
delete [] array;
```

Static vs Dynamic Arrays

Static Array	Dynamic Array
It is created in stack area of memory	It is created in heap area of memory
The size of the array is fixed.	The size of the array is decided during run time.
Memory allocation is done during compilation time.	Memory allocation is done during run time.
They remain in the memory as long as their scope is not over.	They need to be deallocated using delete operator.

Memory Leaks

What will happen if you allocate something and do not deallocate it?

```
void main() {
   int* ptr = new int;
   *ptr=100;
}
```

- A **memory leak** occurs when a piece of dynamic memory (previously allocated by a programmer) is *not properly deallocated* by the programmer
- Even though that memory is no longer in use by the program, it *cannot be used* until it is properly deallocated by the programmer (or the program terminates)

Rule: all allocated memory needs to be deallocated at some point