

# C++ for Image Processing: Pointers, Memory Management and Arrays

## Lecture 2

# Today

- Functions: Parameter passing mechanisms
- Pointers & pointer arithmetic
- Memory management
- Arrays

# Functions

- In Java every function must be an instance method or a static function
- C++ allows functions that are not part of any class
  - think of the `Hello world` program
  - think of the global `main` function

# Parameter passing

- As in Java, function arguments are passed by **value**
  - in C++ object values are not references to actual objects
  - a function receives a copy of the actual argument and the original can never be modified
  - a function receives a copy of the actual argument and can not modify the original
  - recall: in Java methods were able to modify objects
- Solution: **call by value** (like Java) & **call by reference**

# Parameter passing

```
void swap(int &a, int &b)
{ int temp = a;
  a = b;
  b = temp;
}
```

```
int main()
{
  int x, y; x=3; y=7;
  swap(x,y);
  cout << x << ' ' << y;
}
```

> If a parameter is passed **by reference**, the function can modify the original

- indicated by a **&** after the parameter type

> In C++ you always use call by reference when a function needs to modify a parameter

# Parameter Passing

- By value:

```
void swap(int x, int y)
{ int temp = x; x = y; y = temp; }
```

- By reference:

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

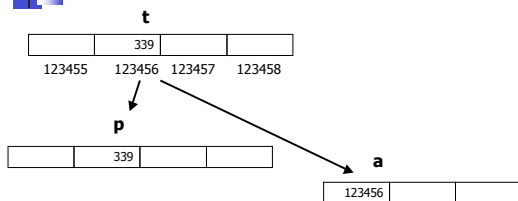
## Pointers: Some basics

- Pointers allow you to directly deal with memory management
- Not something you had to do with Java
- You know the basics about computer memory organisation? [it's beyond our scope today]
- Just remember that memory is split into slots
  - all slots have a **unique number** linked with them
  - e.g. referring to slot #123456 is always referring to the same memory location / address

## Address dereference (&)

- We only mentioned this 2 slides back
- It literally means "the address of"
- Let's consider an example:
  - t is at memory location 123456
  - t = 339
  - p = t
  - a = &t

## Address dereference (&)



t is at address 123456  
t = 339  
p = t  
a = &t

## Pointers

- A variable that holds the address of another variable is a **pointer**
  - a pointer to an *int myInt* refers to the memory address of *myInt*
- How to declare a pointer?
  - place the **star operator** \* between the data type and the variable
  - `int *myIntegerPointer;`

## You must initialise pointers

```
int main() {  
    int myInt = 1000;  
    int *myIntPtr;  
    myIntPtr = &myInt;  
  
    cout << myInt << ' ' <<  
    myIntPtr;  
}
```

What will cout print?

- It is essential to **initialise pointers**
- At declaration the pointer will point at nothing, or at a random memory slot
- Using an uninitialised pointer is guaranteed to cause *segmentation fault* or *bus error* at run time and your program will crash

## How to change the value that a pointer points to

- `myIntPtr=50;`
- What does this line do?
  - sets the value of `myIntPtr` to 50
  - but what is the value?
  - the value is the memory location, so this line will change the location that `myIntPtr` points to, to location number 50
- What if we want to change the integer that `myIntPtr` points to?

## Dereferencing pointers

- `myIntPtr` means: the memory address of `myInt`
- `* myIntPtr` means: the integer at memory address `myIntPtr`
- Use the star operator to **dereference pointers**
  - what was the other use of the star operator?

## Dereferencing pointers

```
int myInt = 1000;
int *myIntPtr = &myInt;

cout << myInt;

*myIntPtr +=5;

cout << myInt;
```



1000  
1005

## What about this?

```
int myInt = 1000;
int *myIntPtr = &myInt;
```

```
int mySecondInt = *myIntPtr;
// this is a second integer whose value
// is that of the integer pointed to by
// myIntPtr
```

```
*myIntPtr +=5;
cout << myInt << '\n'; 1005
cout << mySecondInt; 1000 or 1005?
```

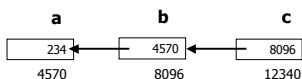
- `mySecondInt` is a wholly new integer at a different memory address
- We created a copy of the value
- We can print the memory addresses of the integers to verify that they are different (i.e. we have a copy)
  - `cout << &myInt;`
  - `cout << &mySecondInt;`

## Many pointers to the same address

- It is possible to have many pointers point to the same address
- Changing the value of the number at that address will change the value the other pointers are pointing to
- Try it out with a simple test program at the lab

## Pointers to pointers

```
int a;
int *b;
int **c;
a = 234;
b = &a;
c = &b;
```



- `c` is a variable of type (`int **`) with a value of 8096
- `*c` is a variable of type (`int*`) with a value of 4570
- `**c` is a variable of type `int` with a value of 234

## Pointers to objects

- So far we only saw pointers to integers
- We can assign pointers to other types, including objects
- In Java you could write
  - `Foo myFooInstance = new Foo(0,0);`
- In C++ you can't: the `new` operator will return a pointer to whatever follows it
  - `Foo *myFooInstance = new Foo(0,0);`



## Pointers to objects

- Once we create a pointer to an object, we can call methods of that object
  - in Java: `myFooInstance.bar()`;
  - in C++: `myFooInstance->bar()`;
  - this dereferences the pointer and calls the method
  - instead of `(*myFooInstance).bar()`;



## Pointers to functions

- You can have pointers to functions
- The greatest use of this is for passing a function as a parameter to a function
- The syntax is awkward:
  - `double (*name) (param_list)`



## Parameter Passing

- By value:
 

```
void swap(int x, int y)
{ int temp = x; x = y; y = temp; }
```
- By reference:
 

```
void swap(int &x, int &y)
{ int temp = x; x = y; y = temp; }
```
- By pointer:
 

```
void swap(int *x, int *y)
{ int temp = *x; *x = *y; *y = temp; }
```



## Pointer arithmetic

- Only +, - make sense
- Depends on the pointer type: the size, in bytes, of the type is added or subtracted
- Big danger here: this is **system/architecture dependent**
  - implementing your code based on this can impact on compatibility



## Example

```
char *pC = "Hello World";
```

```
int *pInt =
reinterpret_cast<int*>(pC);
++pInt;
char *pChar =
reinterpret_cast<char*>(pInt);
cout << *pChar << endl;
```

- What will cout print?
- On 32 bits platform, pointer increment is by 4 bytes
  - cout: "o"
- On 16bit platform by 2 bytes
  - cout: "l"



## Memory management

- In Java you do not have to worry about memory - everything is done for you
- In C++ you have to do things yourself
- Distinguish between
  - local storage**: valid only within a scope - automatic memory, or memory on the **stack**
  - global storage**: valid throughout the execution of the program - free store, dynamic memory, storage on the **heap**



## Memory in global store

- We can request memory in global storage using the **new** keyword
  - `int x;` // creates on stack
  - `int *y = new int(5);` // on heap
  - `object = new classname(params);`
  - on success, memory equal in size to the object created is allocated
- Objects created in this way are valid throughout the execution of the program



## Deallocating memory

- In Java a garbage collector frees memory when no existing objects use it
- In C++ (you guessed) you have to do it
  - use **delete(object)**
  - **REMEMBER:** only objects created using *new* can be deleted with *delete*
  - `Animal *a = new Chicken();`  
`delete a;`
- Forgetting to free memory will cause your program to swell in size



## Memory management in classes

- We mentioned **destructors** in the first lecture, but we did not say anything about them
- Memory that is created with **new** is not deallocated automatically
  - memory allocated in a constructor should be freed using a destructor
  - memory allocated in a function should be freed before the function exits



## Example: no memory leaks

```
class Foo {
private:
    Bar *m_barPtr; //class Bar is defined
                  // somewhere else

public:
    Foo() {}
    ~Foo() {}
    void funcA() { m_barPtr = new Bar; }
    void funcB() { // use object *m_barPtr }
    void funcC() { // ...
                  delete m_barPtr; }
};
```

```
Foo myFoo;
myFoo.funcA();
//....
myFoo.funcB();
//...
myFoo.funcC()
```



## Example: memory leaks etc.

```
class Foo {
private:
    Bar *m_barPtr;
public:
    Foo() {}
    ~Foo() {}
    void funcA() { m_barPtr = new Bar; }
    void funcB() { // use object *m_barPtr }
    void funcC() { // ...
                  delete m_barPtr; }
};
```

Still a **bad example** of memory management

```
Foo myFoo;
//....
myFoo.funcB();
//...
myFoo.funcA()
//...
myFoo.funcA()
//...
myFoo.funcB()
```



## A possible pitfall

- Returning a pointer to an object that is of local scope inside a function
 

```
Foo *badFoo : : createBadFoo(int a, int b) {
    Foo aLocalFooInstance(a,b);
    return &aLocalFooInstance; }
```
- What is wrong in the above code?
- To avoid such problems:
  - **Never** return pointers to variables you did not *new* (unless you are 100% sure they will not leave scope)

## Arrays

- For 1-dimensional arrays
  - `type var_name[size];` (in local storage)
  - `double balance[10];`
  - `balance[7] = 123890.34`
- For 2-dimensional
  - `int mark[10][15];`
  - `mark[3][7] = 234;`
- Index from 0 to (size-1)

## Arrays & Pointers

- Arrays & pointers are closely related
- An array name (with no index) is a **pointer to the first element in the array**
- An alternative way to declare arrays
  - `int *intArray = new int[10];`
  - this will allocate memory in which storage?
  - the value of `intArray` is the address of the first element contained in the array

## A bit more on array indexing

- `intArray[0]` means the integer at memory address `intArray`
- `intArray[1]` means the integer at memory address `intArray + 1`
- `intArray[i]` means the integer at memory address `intArray + i`, equivalent to `*(intArray + i)`

## Deleting arrays

- Memory allocated for an array using `new` needs to be released using `delete`
  - `delete [] pointer_name;`
- Remember to tell the compiler you are deleting the entire array, not just an element
  - `int * pointer_name = new int[7];`  
`delete [] pointer_name;`
  - do not use `delete pointer_name`

## Parameter Passing

- By value:

```
void swap(int x, int y)
{ int temp = x; x = y; y = temp; }
```
- By reference:

```
void swap(int &x, int &y)
{ int temp = x; x = y; y = temp; }
```
- By pointer:

```
void swap(int *x, int *y)
{ int temp = *x; *x = *y; *y = temp; }
```
- Arrays

```
void foo(int someArray[], int length) OR
void foo(int *someArray, int length)
```