

Introduction to Programming using C++

Lecture Four: Pointers & References

Carl Gwilliam

gwilliam@hep.ph.liv.ac.uk

<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

The swap function

“I want to write a function that takes two integers and transposes their values.”

- Swapping the values in the function will have no effect on the variables in **main()**
- Calling a function in this way is referred to as **passing by value**.
- In fact, the object (variable) within the function is actually a **copy** not the original (just has the same label).

```
int main() {  
    int x = 2, y = 5;  
    cout << x << ", " << y << endl;  
    swap(x,y);  
    cout << x << ", " << y << endl;  
}  
  
void swap(int x, int y) {  
    //not same x and y (copies)  
    int tmp = y; y = x; x = tmp;  
    cout << x << ", " << y << endl;  
    return;  
}
```

2, 5

5, 2

2, 5

Is there an alternative method that allows the modification of the input variables?

Overloading in C++

Yes! But you first have to understand the concepts of pointers and references.

- Variable and function names are not important, it is the context in which they are used that matters.

Study these two operators closely:

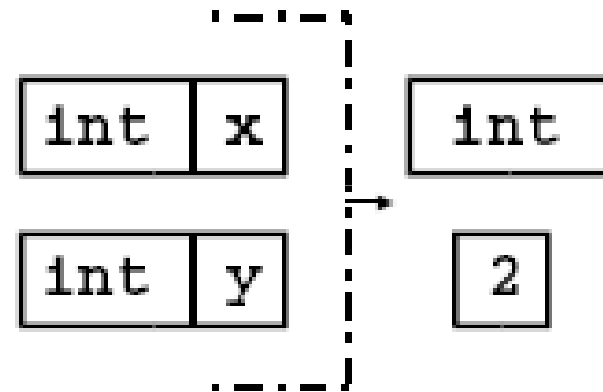
& *

- To understand how pointers and references work you must appreciate that these operators can be overloaded as well.

References

- A reference is just an **alias** for an object.
- References allow multiple variables to access the same object.

```
int x = 2;  
// y is a ref to an int object  
int& y = x;
```



- A reference is initialised by placing an ampersand (&) **between** the type and the variable name.
- A reference must be initialised to an object when it is defined and cannot subsequently reference a diff object

References

- Compare these two pieces of code:

```
int carl = 27;           // assign age to name
int lecturer = carl;    // copy object
carl++;                 // birthday
// display age
cout << "Carl is " << carl << endl;           // Carl is 28
cout << "Lecturer is " << lecturer << endl; // Lecturer is 27
```

```
int carl = 27;           // assign age to name
int& lecturer = carl;    // reference to same object
carl++;                 // birthday
// display age
cout << "Carl is " << carl << endl;           // Carl is 28
cout << "Lecturer is " << lecturer << endl; // Lecturer is 28
```

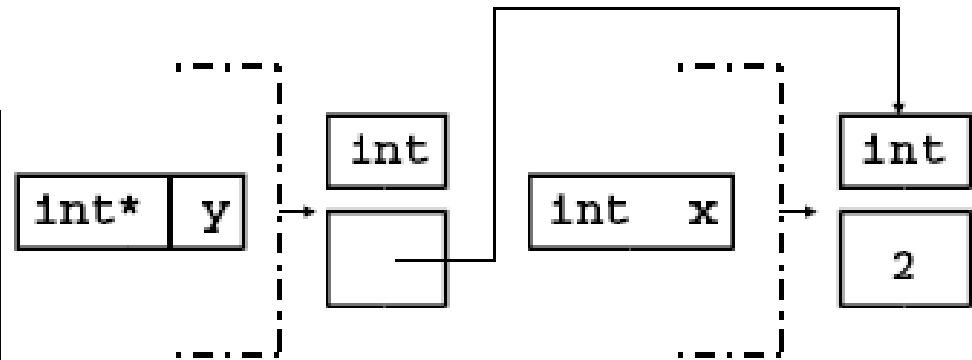


- Reference is changed with original obj, while copy isn't
- Anything done to ref affects original obj and vice versa!

Pointers

- A pointer is an object that points to or refers to another object but, unlike a reference, a pointer is a **new** object. The reference **is** the object.
- The value of the new pointer object is the **address** of the object it is pointing to.

```
int x = 2;  
// y is a ptr to an int object  
int* y = &x;
```



- A pointer is initialised by placing an asterix (*) **between** the type and the variable name.
- Unlike a reference, a pointer can be subsequently changed to point to another object
- Note: An **&** before a variable is the “**address of**” not a reference (operator overloading)

Pointers

```
int carl = 27;           // assign age to name
int* lecturer = &carl;   // pointer to address of object
carl++;                  // birthday
// display age
cout << "Carl is " << carl << endl;
cout << "Lecturer is " << lecturer << endl; // address
cout << "Lecturer is " << *lecturer << endl; // value pointed to
// change lecturer to paul
int paul = ??;
lecturer = &paul;
(*lecturer)++;          // another birthday (only inc obj currently pts to)
// display age again
cout << "Carl is " << carl << endl; // Carl is still 28
cout << "lecturer is " << *lecturer << endl; // Lecturer is 1 year older
```

Carl is 28
lecturer is 0xbffffbb4
lecturer is 28
Carl is 28
lecturer is ??+1

- We get the value of the obj pointed at by a ptr by placing an * before it. This is called **dereferencing** the ptr.
- Note: A * before a ptr is the “**value pointed at**”

Passing by reference

- The input arguments in a function can be modified by passing a reference to the object (i.e. its address), rather than the value (i.e a copy) of the object.
- This is achieved by placing an **&** between the type and variable name in the args of the function to indicate a reference (otherwise syntax is unchanged)
- This method is known as **passing by reference**.

```
void swap(int& x, int& y);  
int main() {  
    int x = 2, y = 5;  
    cout << x << ", " << y << endl;  
    swap(x,y);  
    // still 5,2 as func refs  
    // original variables  
    cout << x << ", " << y << endl;  
}  
void swap(int& x, int& y) {  
    int tmp = y; y = x; x = tmp;  
    cout << x << ", " << y << endl;  
    return;  
}
```

2, 5
2, 5
2, 5

Passing by reference using Pointers

- A pointer to an object (again address) can also be used pass by “reference” into a function.
- To do this we place an * between the type and variable name in the args (and dereference to get obj’s value)

```
void swap(int* x, int* y);
int main() {
    int x = 2, y = 5;
    cout << x << ", " << y << endl;
    swap(&x, &y);           // pass address (pointer)
    cout << x << ", " << y << endl; // still 5,2
}

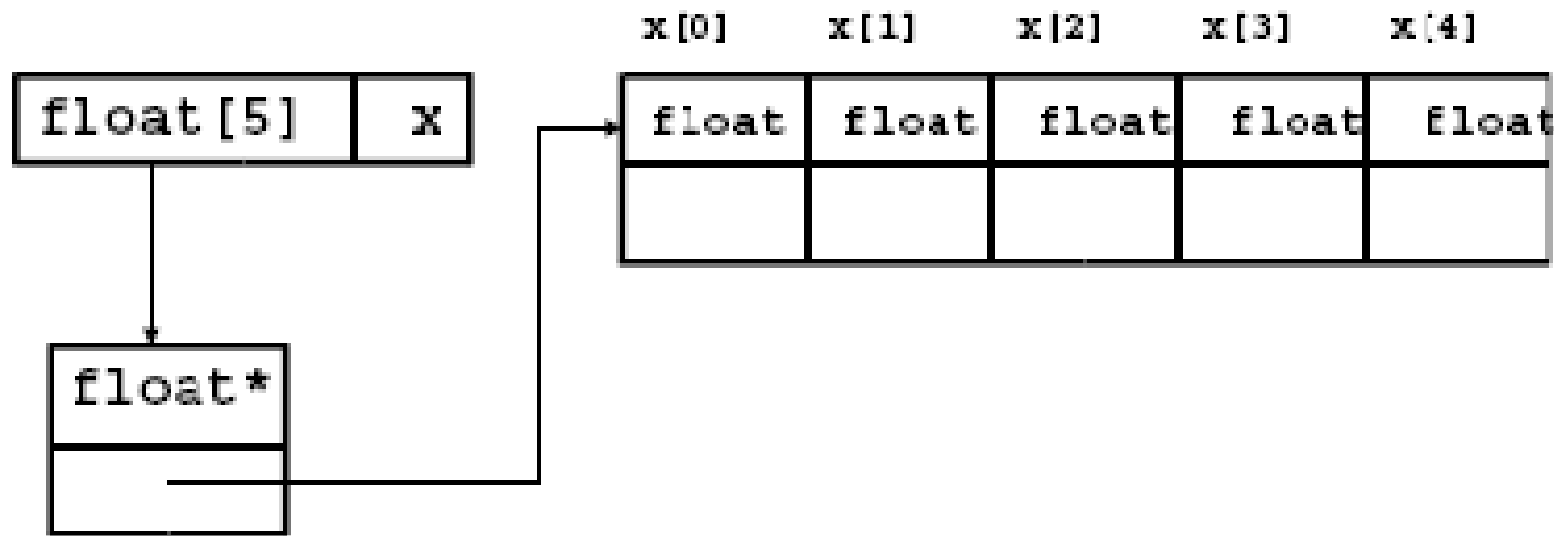
void swap(int* x, int* y) {
    int tmp = *y; // dereference y to get value pointed at
    *y = *x;      // value pointed by y = that pointed by x
    *x = tmp;     // value pointed by x = tmp
    cout << *x << ", " << *y << endl;
    return;
} // aaargggghhhh!!
```

Pointers are too confusing! Why bother with them at all?

But you have already been using pointers in your code before you knew about references...

Pointers and Arrays

- In C++, pointers and arrays are inextricably linked.



- For an array `x[]`, the name `x` acts as a pointer to the first element in the array (i.e. `x[0]`):

```
int x[5] = {1,2,3,4,5};  
cout << "first element of x: " << *x << endl;
```

Pointer arithmetic

- Other elements in the array can be accessed by applying an offset value to the pointer.

`*x` is equivalent to `x[0]`
`*(x+i)` is equivalent to `x[i]`

Must dereference **after** increment

first element of x: 1
third element of x: 3
sixth element of x: ???

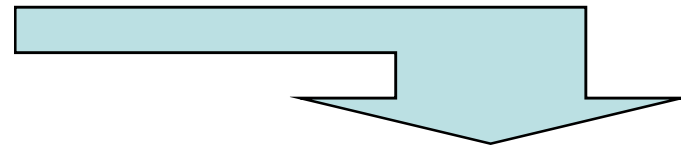
```
int x[5] = {1,2,3,4,5};  
cout << "first element of x: " << *x << endl;    // x[0]  
cout << "third element of x: " << *(x+2) << endl; // x[2]  
cout << "sixth element of x: " << *(x+5) << endl; // x[5]
```

- This is why you have to be careful with arrays! There is no error if you access an address outside of the memory allocated to the array.
- If you write a value to this address you could be overwriting memory used by the operating system...

Pointer arithmetic

- This alternative array notation can be introduced into the sorting algorithm of the **stats** program:

```
// bubble sort algorithm
for (int i = 0; i < (size - 1); i++) {
    for (int j = 0; j < (size - 1); j++) {
        if (sample[j] > sample[j+1]) {
            float swap = sample[j];
            sample[j] = sample[j+1];
            sample[j+1] = swap;
        }
    }
}
```



```
// bubble sort algorithm
for (int i = 0; i < (size - 1); i++) {
    for (int j = 0; j < (size - 1); j++) {
        if ( *(sample + j) > *(sample + j+1) ) {
            float swap = *(sample + j);
            *(sample + j) = *(sample + j+1);
            *(sample + j+1) = swap;
        }
    }
}
```

- Remember the difference between ***(x++)** and **(*x)++**

Returning pointers from functions

- So far, there has been a restriction of one return value per function call. What if an array needs to be returned?
- A pointer is a single object that can be used to indicate the first element in an array.
- A pointer to an array can be returned from the function. The array can then be accessed through the pointer.

```
float* x = getarray();  
float* getarray() {  
    int* x = new int[2];  
    x[0] = 1; x[1] = 2;  
    //array converted  
    //to pointer to float  
    return x;  
}
```

- You can't create the array normally as it'll only be valid within the function's scope.
- This means it'll be deleted at the end of the function and the ptr will point to nothing
- You need to use **new** (later)

Returning pointers from functions

- An array is converted to a pointer as soon as passed into a function. The array contents are therefore passed by “reference” into the function.
- Any modifications of the array contents within the function will be persistent (changes original too). This means there is no need to return the array.
- For example, the contents of the **sample** array in the **stats** program will be arranged in order even after the call to the **getOrder** function has been made.

```
float sample[size];  
void getOrder(float* sample, int size) {  
    // bubble sort algorithm  
    :  
    return;  
}
```

This called returning by reference!

Constants

- During the last two tutorials, you have encountered variables that will always have the same value:

```
double PI = 3.14157;  
double G = 6.67e-11;
```

- These variables only have to be declared and initialised once and never redefined. They are **constants** and are declared by the **const** keyword

```
const double PI = 3.14157;  
PI = 3.2;           // error
```

- It is useful to declare a **const** variable as a global variable if it is called throughout the program.

Constant pointers and references

- Unlike references, pointers do not always refer to the same object. Pointers can be reassigned.
- Pointers can be fixed to one object (i.e. address) using **const**.
- Pointer to a constant value => Value it points to can't be changed but pointer can be reassigned to a diff address.
- Constant pointer to a value => Value may be changed but pointer can't be reassigned to a different address.
- Const pointer to const value => Value can't be changed **and** pointer can't be reassigned a different address.

```
const double* pPI = &PI;  
pPI = &x;           // ok
```

```
double* const px = &x;  
px = &PI;           // error
```

```
const double* const cpPI = &PI;  
cpPI = &x           // error
```


Some questions about functions

- Can my functions be used by other programs?
 - Yes. Place your functions in a separate translation unit.
- How do I supply a default value for a function argument?
 - Include default value for the argument in the function prototype.
- Can a function call itself?
 - Yes, these are known as recursive functions.
- Do I have to write separate functions for each type, even though the function will be exactly the same?
 - In general, yes. But these functions can have the same identifier (this is known as **overloading** a function).

Some questions about functions

- Is it possible to call a function that modifies the input arguments?
 - Yes. Instead of passing the **value** of the variables to the function you pass the **references** of these variables.
- How do I return more than one value from a function?
 - Return a **pointer** to an array storing the values or include the array as an input argument in the function.
- Can the value of a function variable be made constant throughout the entire program?
 - Yes, use the const keyword when variable is declared. The const keyword can also be used to assign pointers to a fixed address.