Pointers and References

Steven Zeil

October 2, 2013

Contents

1 References **2**

2 Pointers **10**
2.1 Working with Pointers . . . 11
2.1.1 Memory and C++ Programs . . . 15
2.1.2 Allocating Data . . . 20
2.2 Pointers Can Be Dangerous . . . 23

3 The Secret World of Pointers **25**
3.1 Pointers and Arrays . . . 25
3.2 Pointer and Strings . . . 28
3.3 Pointers and Member Functions . . . 30

In this lesson, we examine the data types that C++ provides for storing addresses of data.
These are important because they

- Allow access to objects created "on the fly" (dynamic storage allocation)
- Permit access to "large" objects without copying
- Allow selective access to parts of large structure

Indirection

- Pointers and references allow us to add a layer of “indirection” to our data.
- Instead of giving an answer directly, we give the location where the answer can be found.

. . . . . .

It’s rather like answering a question about, say, the meaning of the word “Ragnarok” by pointing to a nearby dictionary instead of explaining it directly.

In our code, sometimes we will employ this indirection for efficiency. In other cases we use it for flexibility or to simplify our code.

A pointer or reference is the "name" or "address" of an object NOT the object itself.

Most variables have names assigned to them by the programmer at the time the program is written (e.g. `"int numberOfCourses;` Every variable in a program has an machine address where that variable is stored in the main memory of the computer. Think of this as the "machine’s name" for the variable.

Every variable in a program has a value which is the data stored in the variable’s address.

1 References

References

Reference types are introduced by the use of “&” in a type expression.

- Reference types hold the address of an already existing data value

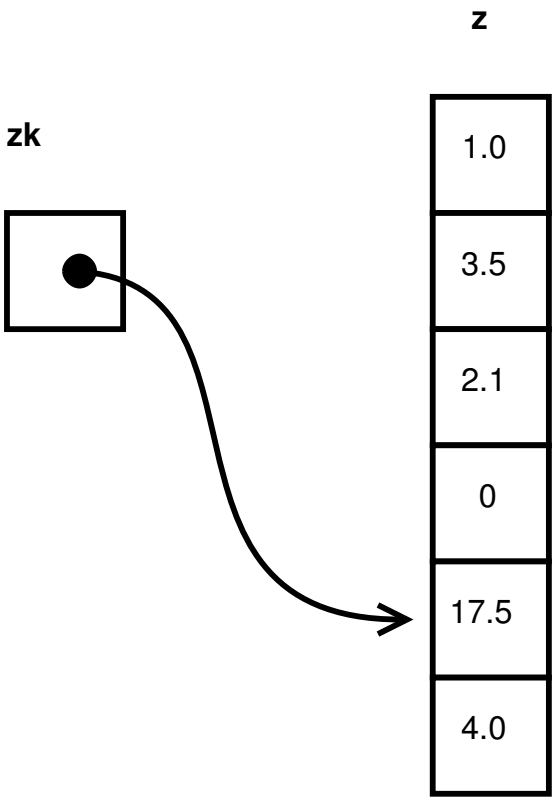
Example

```
double z[1000];
   ⋮
int k = i + 200*j;
double& zk = z[k];
```

zk holds the address of *z[k]*.

. . . . . .

Initializing References



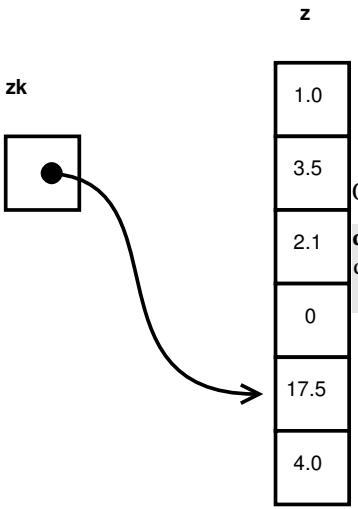
When reference variables are declared, they must be immediately initialized to the location of some existing data value, e.g.,

```
double& zk = z[k];   // zk get address of z[k]
```

One initialized, a reference *cannot* be reset to point to a different location

.....

Accessing data Via References



One initialized, we can use the reference much like any ordinary variable:

```
double& zk = z[k];   // zk get address of z[k]
cout << zk; // accesses data stored
            // at that location
```

. . . . . .

Assignment and References

Subsequent assignments to a reference variable will store new values at that location, but will not change the location.

```
double& zk = z[k];   // zk get address of z[k]
zk = 1.0; // changes the value of z[k]
++k;
zk = 2.0; // changes the value of z[k-1]
```

. . . . . .

Example: working with indirection

Question: What would the output of the following code be?

```
int a = 1;
int b = a;
cout << a << " " << b << endl;
```


```
a = 2;
cout << a << " " << b << endl;
b = 3;
cout << a << " " << b << endl;
```

.....

Answer:

```
1 1
2 1
2 3
```

a and *b* are distinct variables. A change to one has no effect on the other.

. . . . . .

Example: working with indirection (cont.)

Let's make a one-character change. . .

Question: What would the output of the following code be?

```
int a = 1;
int& b = a;
cout << a << " " << b << endl;
a = 2;
cout << a << " " << b << endl;
b = 3;
cout << a << " " << b << endl;
```

. . . . . .

Answer:

```
1 1
2 2
3 3
```

a and *b* are "synonyms" for the same data value. A change to one can be seen via the other.

. . . . . .

References and Loops

Once initialized, a reference cannot be reset to a different location.

- However, if we declare a reference within a loop body:

```
for (int i = 0; i < 100; ++i)
{
   int k = longAndComplicatedCalculation(i);
   double& zk = z[k];
   doSomethingWith (zk);
}
```

  Then each time around the loop we are initializing a *different* reference variable

  - Variables that are declared inside a { } are destroyed when we reach the closing }.
  - So, each time around the loop, *zk* refers to a different element of the array.

. . . . . .

The above example suggests one reason why we may use references – to avoid repeating long and complicated calculations to select array elements or struct members.

Const References

When we modify a reference type by pre-pending "const":

- We are allowed to look at the data value whose address is stored in the reference.

- But we cannot alter the data value via that reference

```
Money price (24, 95);
Money& salePrice = price;
const Money& oldPrice = price;
price.dollars = 25; // OK
salePrice.dollars = 26; // OK
oldPrice.cents = 0; // illegal, cannot change value
```

- Note that both of the legal assignments would affect the values seen in all three variables.

. . . . . .

References and Functions

You've seen lots of functions using reference parameters

```
void foo (Money& m, const Time& t);
```

- We've explained in the past that this was "pass by reference", a special parameter passing mechanism.
- In fact, as far as the compiler is concerned, there is no special parameter passing mechanism.
  - This is still "pass by copy".
  - But the type of data being copied and passed are references
  - All the "special" behavior associated with reference parameters stems from the normal properties of references
    - Passes addresses instead of actual data
    - Allows alteration of data back in the caller...
    - Unless the reference type is marked `const`, in which case we can "look but not touch".

. . . . . .

References and Functions (returns)

References are also sometimes used as return types

```
const Money& getAmountBid (Bid& b)
{
   return b.amountBid;
}
   ⋮
cout << getAmountBid(myBid).dollars;
```

Slight improvement in efficiency – the *Money* value does not need to be copied.

. . . . . .

References and Functions (returns) cont.

```
Money& getAmountBid (Bid& b)
{
   return b.amountBid;
}
   ⋮
getAmountBid(myBid).dollars = 0;
```

. . . . . .

2 Pointers

Pointers

Pointers, like references, store the location or address of data.

- Pointers relax many of the restrictions on references
  - Pointers need not be initialized when declared (a bad idea)
  - When initialized, they need not hold an address of existing data

    - can be null instead
  - After initialization, they can be reassigned to refer to a different data element.

- But pointers lose some of the convenience of references:
  - Require special syntax to access the data referred to by a pointer

. . . . . .

2.1 Working with Pointers

Declaring Pointer Variables

A pointer declared like this

```
double *p;
```

contains, essentially, random bits.

To be useful, it must be initialized

```
double *p = ...
```

or, later, re-assigned

```
p = ...
```

. . . . . .

Initializing Pointer Variables

Where do new pointer values come from?

- Pointers can be given the addresses stored in other pointers:

```
int* q = ...
  ⋮
int* p = q;
```


  - But that doesn't really answer the question. It just defers it.

- Pointers can be assigned the address of an existing data value via the & operator:

```
int a = 23;
int *p = &a;
```

  - But this is rare, dangerous, and generally frowned upon

- Pointers can be given the address of newly allocated data:

```
int *p = new int;
int *pa = new int[100];
```

- Pointers can be set to *null*, a special value indicating that it does not point to any real data.

  There are multiple ways to do this:

  - "0" is a null pointer

```
int *p = 0;
```

    * That's not zero. It's a null pointer.

  - *NULL* is a special symbol declared in `<cstdlib>`

```
#include <cstdlib>
   ⋮
int* p = NULL;
```

  - Coming soon, courtesy of the new C++11 standard

```
int *p = nullptr;
```

.....

NULL is actually a bit of a problem. Not only do you have to include a special header to get it, but there are some rare circumstances where passing it to functions that take a pointer as parameter will not compile properly. Hence the new standard introduced a better-behaved universal null pointer constant.

This won't be available, however, until compilers take the C++11 features out of their beta status.

Dereferencing a Pointer

Accessing data whose address is stored in a pointer is called *dereferencing* the pointer.

- The unary operator * provides access to the data whose location is stored in a pointer:

```
Money *p = new Money(100, 25);
   ⋮
Money m = *p;       // * gets the whole data element
*p = Money(0,15); // and we can store there
```

.....

Dereferencing and Structs

- The operator -> provides access to struct members via a pointer:

```
Money *p = new Money(100, 25);
   ⋮
int totalCents = 100*p->dollars + p->cents;
p->dollars = 0;
```

  - These two statements are equivalent:

```
p->dollars = 0;
(*p).dollars = 0;
```

.....

Assignment and Pointers

Subsequent assignments to a pointer variable will change the location it points to.

```
double* zk = &(z[k]);   // zk get address of z[k]
*zk = 1.0; // changes the value of z[k]
zk = &(z[k+1]);
zk = 2.0; // changes the value of z[k+1]
```

.....

Example: working with pointers

Question: What would the output of the following code be?

```
int a = 1;
int b = 2;
int* pa = &a;
int* pb = &b;
cout << a << " " << *pa << " " << b << endl;
a = 3;
cout << a << " " << *pa << " " << b << endl;
*pb = 4;
cout << a << " " << *pa << " " << b << endl;
pa = pb;
cout << a << " " << *pa << " " << b << endl;
```

.....

Answer:

```
1 1 2
3 3 2
3 3 4
3 4 4
```

. . . . . .

Const Pointers

When we modify a pointer type by pre-pending "const":

- We are allowed to look at the data value whose address is stored in the reference.
- But we cannot alter the data value via that reference

```
Money price (24, 95);
Money* salePrice = &price;
const Money* oldPrice = salePrice;
price.dollars = 25; // OK
salePrice->dollars = 26; // OK
oldPrice->cents = 0; // illegal, cannot change value
```

- Same as the affect of `const` on references

. . . . . .

2.1.1 Memory and C++ Programs

Where is data Stored?

The memory of a running C++ program is divided into three main areas:

- The *static area* holds variables that have a single fixed address for the lifetime of the execution.

  - Variables declared outside of any enclosing `{ }` or marked as `static`.

- The *runtime stack* (a.k.a. *activation stack* or *automatic storage*) has a block of storage for each function that has been called but from which we have not yet returned.
  - All copy parameters and local variables for the function are stored in that block.
  - The block is created when we enter the function body
  - and destroyed when we leave the body.
- The *heap* is a programmer-controlled "scratch pad" where we can store variables
  - But the programmer has the responsibility for managing data stored there

.....

How Functions Work

```
int foo(int a, int b)
{
  return a+b-1;
}
```

would compile into a block of code equivalent to

```
  stack[1] = stack[3] + stack[2] - 1;
  jump to address in stack[0]
```

.....

The Runtime Stack

- the "stack" is the *runtime stack* (a.k.a. the *activation stack*) used to track function calls at the system level,
- stack[0] is the top value on the stack,
- stack[1] the value just under that one, and so on.

.....

An Example of Function Activation

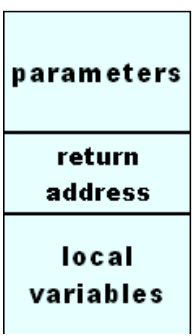
Suppose that we were executing this code, and had just come to the call to resolveAuction within `main`.

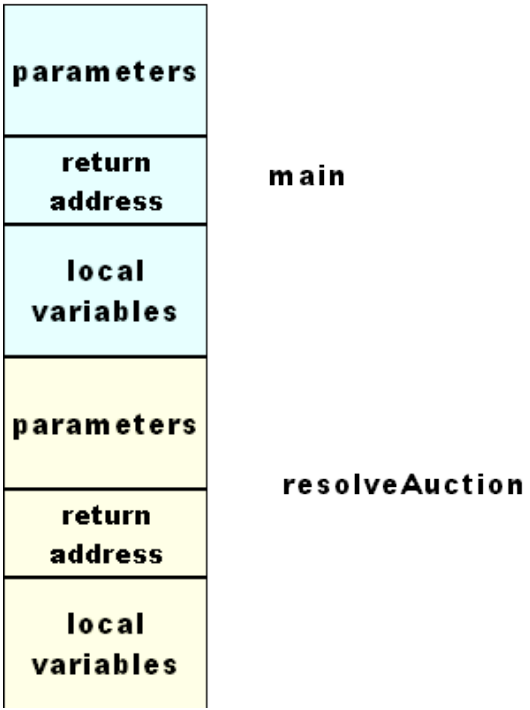
```
#include "time.h"

void resolveAuction (Item item)
{
  ⋮
  int h = item.auctionEndsAt.getTime();
  ⋮
}

int main (int argc, char** argv)
{
  ⋮
  resolveAuction (item);
  ⋮
}
```

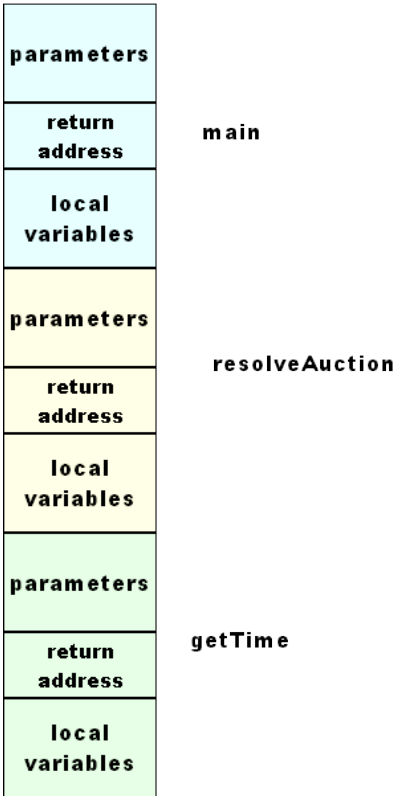
The runtime stack (a.ka., the activation stack) would, at this point in time, contain a single *activation record* for the `main` function, as that is the only function currently executing:

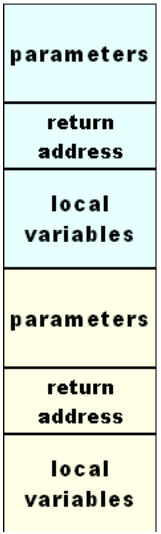




When `main` calls `resolveAuction`, a new record is added to the stack with space for all the data required for this new function call.

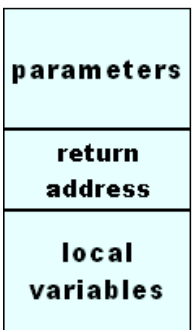
When `resolveAuction` calls `getTime`, another new record is added to the stack with space for all the data required for *that* new function call.





But once `getHours` returns (to `resolveAuction`), that activation record is removed from the stack. `resolveAuction` is once again the active function.

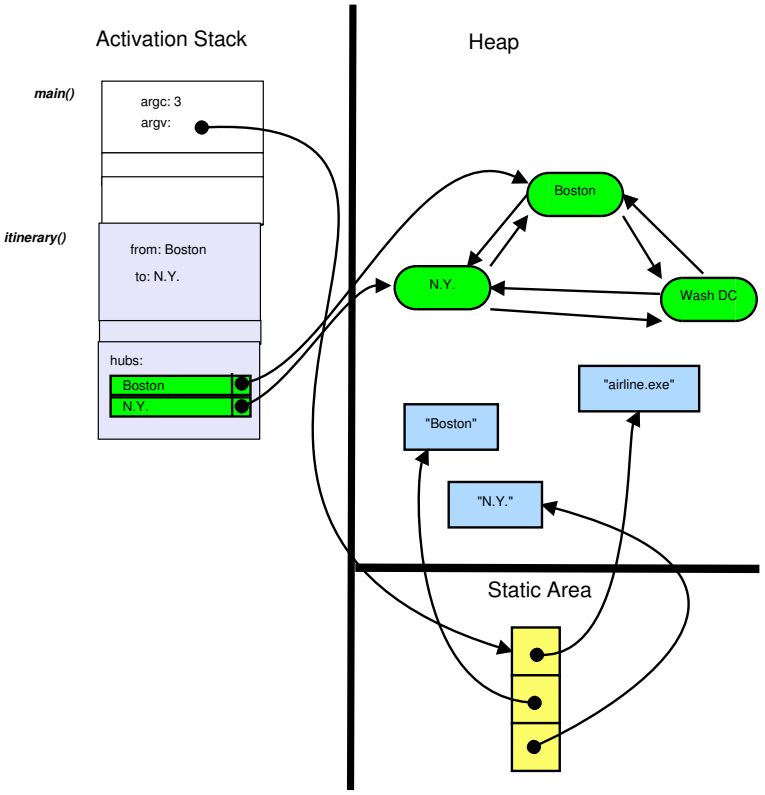
And when `resolveAuction` returns, its record is likewise removed from the stack.



.....

2.1.2 Allocating Data

Example of Overall Memory layout



Allocating Data on the Heap

We allocate data with `new` and remove it with `delete`:

```
int *p = new int;
int *pa = new int[100];
```

```
    ⋮
delete p;
delete [] pa;
```

Note the slightly different forms for arrays versus single instances.

. . . . . .

Dynamic Allocation

Programmers often distinguish between "dynamic" and "static" activities:

- Something is "dynamic" if it happens at run-time, under control of the program.
- Something is static if it happens at compile-time and/or is controlled by the compiler.

```
int *p = new int;
int *pa = new int[100];
```

Allocation of data via `new` is called *dynamic allocation* of data.

. . . . . .

Dynamically allocated memory is controlled through the two operators *new* and *delete*

- The `new` operator allocates sufficient memory to store one or more objects of the specified type.
  - The type of the object to be allocated follows the `new` operator
  - Memory is allocated from the heap.
- The `delete` operator returns the memory allocated to an object back to the memory pool to be reused.

Summary: Pointers versus References

	References	Pointers
Type Declaration	&	*
Initialization	must be initialized points to existing data	optional may be null
Dereferencing	automatic	*, ->
Management	automatic	new, delete
Dangerous?	minimal	very

. . . . . .

2.2 Pointers Can Be Dangerous

Because pointers provide access a memory location and because data and executable code exist in memory together, misuses of pointers can lead to both bizarre effects and very subtle errors.

Potential Problems with Pointers

- *uninitialized pointers*,
- *memory leaks* and
- *dangling pointers.*

. . . . . .

Uninitialized pointers

- Uninitialized pointer pose a significant thread.
  - the value stored in an uninitialized pointer could be randomly pointing anywhere in memory.
  - Storing a value using an uninitialized pointer has the potential to overwrite anything in your program, including your program itself

- Your best defense:

  Never write a declaration like

```
Money* p;
```

  Always give your pointers an initial value

```
Money* p = 0;
```

  - Null if you can't make it point to a real data value.

. . . . . .

Memory Leaks

A *memory leak* occurs when all pointers to a value allocated on the heap has been lost, e.g.,

```
int isqrt (int i)
{
   int* work = new int;
   *work = i;
   while ((*work) * (*work) > i)
     -- (*work);
   return *work;
}
```

- When we return from this function the local variable *work* is lost.
  - But that has the only copy of the address of the *int* that we allocated on the heap.
- Each call to this function will leak a bit of memory.

. . . . . .

Over time, memory leaks can cause programs to slow down and, eventually, crash.

Worse, a leaky program may come to take up so much of a systems memory that it interferes with the operation of other programs on the same system.

Dangling Pointers

Dangling pointers refer to a pointer which was pointing at an object that has been deleted.

```
int* p = new int;
int* q = p;
  ⋮
delete p;
```

- The pointer *q* still has the address of the object even though the memory for that object has been returned to the system.
- If the memory allocated to the deleted object is re-used for another purpose,
  - The value visible via *q* may appear to "spontaneously" change
  - Storing a value via *q* may corrupt that other data

. . . . . .

3 The Secret World of Pointers

3.1 Pointers and Arrays

What's in a Name? (of an array)

```
int a[100];
double b[1000];
```

- You know what expressions like `a[i]` and `b[2]` do
- But what about just "`a`" or "`b`"?

. . . . . .

Arrays are Pointers

```
int a[100];
double b[1000];
```

- Arrays are really pointers
  - *a* has type `*int`
  - *b* has type `*double`
- They point to the first (`[0]`) element.

.....

You may have observed examples of passing arrays to functions as parameters. They are usually passed as pointers. That's possible because of the fact that arrays really are pointers.

Pointer Arithmetic

- You can add integers to pointers

```
#include <iostream>

using namespace std;

int main()
{
  int *i = new int;
  double *d = new double;
  cout << "i " << i << "  d " << d << endl;
  i = i + 1;
  ++d;
  cout << "i " << i << "  d " << d << endl;
  return 0;
}
```


```
> g++ pointerArith.cpp
> ./a.out
i 0x1eea010  d 0x1eea030
i 0x1eea014  d 0x1eea038
>
```

  - Actually adds to the address the number of bytes required to store one data value of the type pointed to

.....

Pointer Arithmetic 2

- You can subtract pointers from one another

```
#include <iostream>

using namespace std;

int main()
{
  int array[5];
  int* p = &(array[0]);
  int* q = &(array[3]);
  cout << "p " << p;
  cout << "  q " << q;
  cout << "  q-p " << (q-p);
  cout << endl;
}
```

```
> g++ pointerArith2.cpp
> ./a.out
p 0x7fff2065c380  q 0x7fff2065c38c  q-p 3
>
```


– Computes how many element-sized blocks away from wach other the two addresses are

.....

OK, Why do Pointer Arithmetic?

Pointer arithmetic is actually illegal and pretty much useless *except* when the addresses are all within a single array.

```
double b[1000];
```

- *b* is a pointer to the start of the array
- `b[i]` is simply a convenient shorthand for `*(b+i)`

.....

Pointers, Arrays, and Functions

This is why, when arrays are passed to functions, they are generally passed as pointers:

```
double sumOverArray (double* a, int n)
{
   double s = 0.0;
   for (int i = 0; i < n; ++i)
      s += a[i];
   return s;
}
```

.....

We'll have more on this shortly when we look at dynamically allocated arrays.

3.2 Pointer and Strings

Not all strings are created equal.

- In C++'s parent language, C, there is no *std::string*

- Strings were stored in character arrays
  - End of a string was indicated by a byte containing the ASCII character NUL (value 0)
- Less than satisfactory
  - NUL is actually a useful character in some contexts
  - Many string operations were needlessly slow
- C++ retains the C string operations for backwards compatibility.

. . . . . .

String Literals

The most common place where strings and character arrays meet is in string literals.

- `"abc"` does *not* have type *std::string*
- Its data type is actually *const char**
  - And it actually takes up 4 characters, not 3
- The *std::string* type provides a constructor

```
string (const char* charArray);
```

for building new *string*s from character arrays.

. . . . . .

main

The main function in C++ programs has the prototype

```
int main (int argc, char** argv)
```

- *argc* is the number of command line parameters.
- *argv* holds the command line parameters.

Question: Why are there two asterisks in *char***?

.....

The first '*' indicates that each parameter is a C-style character array (passed, as is common for arrays, as a pointer).

The second '*' indicates that this is an array of those character arrays (again, passed as a pointer), because there can be multiple command line parameters.

3.3 Pointers and Member Functions

Hide the Parameter

Remember that when we convert standalone functions to member functions, one parameter becomes implicit:

```
struct Money {
   ⋮
};
Money add (Money left, Money right);
```

becomes

```
struct Money {
   ⋮
   Money add (Money right);
};
```

.....

Revealing the Hidden Parameter

That parameter really does exist

- Its name is "*this*"
- Its data type, for any struct *S*, is S*

```
struct Money {
   smvdots
   Money add (/* Money* this, */ Money right);
};
```

. . . . . .

Using this

Sometimes we need to make explicit reference to the implicit parameter.
Suppose that we had

```
Money Money::add (Money right)
{
  Money result;
  result.dollars = dollars + right.dollars;
  result.cents = cents + right.cents;
  return result;
}
```

and wanted to add some debugging output...

. . . . . .

Explicit this

```
Money Money::add (Money right)
{
  cerr << "Entering Money::add, adding "
       <<  *this
```


```
        << " to " << right << endl;
  Money result;
  result.dollars = dollars + right.dollars;
  result.cents = cents + right.cents;
  return result;
}
```

There's really no other way to pass the whole "left" value to another function or operator.

. . . . . .

The need to explicitly refer to *this* is unusual, but not all that rare.