

## Re: Why "const rect& rhs" is used here?

---

*Source:* <http://www.tech-archive.net/Archive/VC/microsoft.public.vc.language/2006-09/msg00444.html>

---

- *From:* "Alan Carre" <[alan@xxxxxxxxxxxxxxxxxxxxx](mailto:alan@xxxxxxxxxxxxxxxxxxxxx)>
  - *Date:* Mon, 18 Sep 2006 07:54:46 +0700
- 

"Jacky Luk" <[jl@xxxxxxxxxxx](mailto:jl@xxxxxxxxxxx)> wrote in message  
[news:uQu0kLx1GHA.1252@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx](mailto:news:uQu0kLx1GHA.1252@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx)

Hi dudes,

```
class rect
{
friend ostream& operator<<(ostream& os, const rect& rhs);
public:
rect (int w, int h) : _w(w), _h(h)
{ _area = _w * _h; }
bool operator < (const rect& rhs) const <<<< this one
{ return _area < rhs._area; }
private:
int _w, _h, _area;
};
```

=====

how come (const rect& rhs) is used  
as far as i can see, the rhs is not mutated inside operator < and not  
necessary to be called by reference because  
rhs will never be changed. And how come the operator function needs to be  
declared as const (on the end of the line), as far as i can see, i also  
don't see why there is a need for that... Any comments?

I thought I would throw in my 2 cents worth here because I think your  
question implies something I've suspected for quite some time but never  
really articulated. And that is., that the 'const' modifier, I think, is  
one of the most misunderstood keywords in the C++ language.

For those of you who've never had the misfortune to participate in coding a  
major project using only straight 'C' I can see how this can come about. So  
let me show you what the world was like back in those days...

First we had functions, and built in types and we had something called a  
"structure". There were no classes., no operators, no  
constructors/destructors, no public/private or anything like that. A  
structure was, or is, what has evolved into the modern concept of the  
'class'. Programming was code-centric and there was no tight binding between

## Re: Why "const rect& rhs" is used here?

code and data as there is today. As well, everything was exposed. Here's an example:

```
enum COURSES
{
    ENGLISH,
    MATH,
    ...
    MAX_COURSES
};

struct STUDENT_RECORD
{
    char pcStudentFirstName[64];
    char pcStudentLastName[64];
    int iGrades[MAX_COURSES];
    int iSkippedClasses;
    int iCurrGrade;
    BOOL bSomeBool;
    // and so on.
};
```

Or something like that (assuming you're writing a program to keep track of students). With no user defined types things got messy very quickly; you had to use `sprintf` to set strings and so on.

So, supposing you had a `STUDENT_RECORD` for a particular student, and you wanted to know what their grade was for math then you'd write down:

```
iMathGrade = record.iGrades[MATH];
```

If you had an array of records and wanted to find out the average math grade over those records (ie. students) you'd do something like:.

```
for (i=0;i<num_students;i++)
{
    iTotMathGrades += records[i].iGrades[MATH];
}

iAverageMathGrade = iTotMathGrades / num_students;
//maybe print it out...
```

So here's the problem. As the structures got more complex, the algorithms for manipulating them also got more complex. For example, our `STUDENT_RECORD` structure contains an array 'iGrades', for each possible course. To calculate the average grade over all courses you'd have to write something like this:

```
iTotalGade = 0;
for (i=0;i<MAX_CLASSES;i++)
{
```

Re: Why "const rect& rhs" is used here?

Re: Why "const rect& rhs" is used here?

```
iTotalGade += record[currStudent].iGrades[i];  
}
```

```
iAverageGrade = iTotalGrade / MAX_CLASSES;
```

But of course a given student probably isn't taking every single course on offer, so you'd probably have another list of BOOLS describing whether or not they had taken the course. So you'd get something like this instead:

```
iTotalGade = 0;  
iTotalCourses = 0;  
for (i=0;i<MAX_CLASSES;i++)  
{  
if (record[currStudent].bHasTakenCourse[i])  
{  
iTotalCourses++;  
iTotalGrade += record[currStudent].iGrades[i];  
}  
}
```

```
// assuming they're taking at least 1 course!  
iAverageGrade = iTotalGrade / iTotalClasses;
```

```
=====
```

Well you can see how this sort of thing could balloon into a great big mess of loops and comparisons and sums. So the idea then was to generate functions which would take the structs as arguments as follows:

```
// Example where struct is copied.  
int AverageGrade (STUDENT_RECORD record)  
{  
int iTotalGade = 0;  
int iTotalClasses = 0;  
for (i=0;i<MAX_CLASSES;i++)  
{  
if (record[currStudent].bIsTakingClass[i])  
{  
iTotalClasses++;  
iTotalGade += record[currStudent].iGrades[i];  
}  
}  
  
return iTotalGrade / iTotalClasses;  
}
```

So then in your main code you might have:

```
int iAverageGradeForAllStudents = 0;  
for (iStudent=0;iStudent<MAX_STUDENTS;iStudent++)  
{
```

Re: Why "const rect& rhs" is used here?

## Re: Why "const rect& rhs" is used here?

```
iAverageGradeForAllStudents += AverageGrade (student[iStudent]);  
}
```

etc...

You can see where this is going. Pretty soon what you wind up with is a structure and then a whole host of functions for creating, manipulating and testing the structures. This has the immediate advantage that you can lay down break points \*in the functions\* to see how the structures are being used. So instead of someone saying

```
strcpy (record.pcStudentFirstName, "Johnny");
```

You'd have a function called SetStudentName:

```
void SetStudentName (STUDENT_RECORD* pRecord, const char* cszName)  
{  
  ASSERT (strlen (cszName) < 64);  
  strcpy (pRecord->pcStudentFirstName, cszName);  
}
```

IMPORTANT\*\*\* Notice that the \*pointer\* parameter cszName is passed in with the "const" modifier. What that means is that the function SetStudentName will not modify the contents of what cszName points to. The STUDENT\_RECORD \*pointer\* parameter (pRecord) however, is not being passed in with a "const" modifier and that means that the function is allowed, even compelled, to modify pRecord. In fact, obviously that is the purpose of this function: to modify the contents of pRecord.

What about this function:

```
void GetStudentName (const STUDENT_RECORD* pcRecord,  
char* szNameOut,  
size_t cntBuffSize)  
{  
  szNameOut[cntBuffSize - 1] = '\0';  
  strncpy (szNameOut, pcRecord->pcName, cntBuffSize - 1);  
}
```

IMPORTANT\*\*\* In this case the function will not modify what is pointed to by pcRecord, but \*will\* modify what is pointed to by szNameOut.

It should be noted however, that C is very flexible and it is possible to get around the const modifier by simple casting. So even though we passed pcRecord in as 'const' there is no \*real\* gurantee that it will not be modified. It is up to the programmer to program responsibly and to follow the guidelines of the function's prototype. In this prototype there is an explicit agreement between the caller and the callee something like this:

Caller

1. I, the caller, am lending you the callee 'GetStudentName' a pointer to a

Re: Why "const rect& rhs" is used here?

Re: Why "const rect& rhs" is used here?

STUDENT\_RECORD structure for the duration of the function call on condition that you, the callee, make no modifications to the data pointed to by pcRecord. I am further passing you a pointer to a memory block of size cntBuffSize starting at location szNameOut which you may modify in any way you see fit.

Callee

2. I, the callee, agree to make no changes whatsoever to the data pointed to by pcRecord and further agree that whatever data I write out during the function call will be within the address range [szNameOut, szNameOut + cntBuffSize - 1]. I shall not be held responsible if the address range [szNameOut, szNameOut + cntBuffSize - 1] overlaps the address range containing pcRecord and hereby declare that such input fall under the category of "undefined behaviour".

This is pretty much the standard agreement between caller and callee and one should always keep it in mind when prototyping or examining the prototype of a function.

=====

So now to get the the point of all this.

Problem 1: The structure definitions must be visible to any module that uses the functions they refer to.

Problem 2: There's still nothing stopping someone from directly modifying the structures should they get a hold on one. (ie. there's still nothing stopping someone from writing down: "strcpy (record.pcStudentName, "Johnny)").

We would like to force people to use the function wrappers Get/Set student name for instance, but how can we do that when everything is exposed?

There are several ways to deal with these (related) problems. One of the most common solutions is to simply not publish the structures at all and ONLY publish the function wrappers. Here is how it's done:

```
////////////////////////////////////
// Student Record.h
void* CreateStudentRecord (void);
void SetStudentName (void* hRecord, const char* pcName);
void GetStudentName (const void* hcRecord, char* pcOut, size_t nBuffSize);
```

and so on.

Now onto the 'C' file:

```
////////////////////////////////////
// Student Record.cpp
struct STUDENT_RECORD
{
```

Re: Why "const rect& rhs" is used here?

```
char pcStudentFirstName[64];
char pcStudentLastName[64];

BOOL bHasTakenCourse[MAX_COURCES];
int iGrades[MAX_COURCES];

int iCurrGrade;
int iSkippedClasses;
BOOL bSomeBool;
// etc
};

void* CreateStudentRecord ()
{
void* hNewRecord = malloc (sizeof (STUDENT_RECORD));
ASSERT (hNewRecord);
return hNewRecord;
}

void SetStudentName (void* hRecord, const char* pcName)
{
STUDENT_RECORD* pRecord = (STUDENT_RECORD*)hRecord;
ASSERT (strlen (cszName) < 64);
strcpy (pRecord->pcStudentFirstName, cszName);
}
```

If you are familiar with Windows programming then this should be second nature to you. In Windows the "void\*"s are called HANDLES. They may not be bare pointers, but they are indices of one sort or another into some struct somewhere in the operating system.

So that solves the problem of data hiding and many others as well. For instance, if you decided to change the STUDENT\_RECORD struct, you need not change any code that uses the handles. The function implementations may change but from the caller's perspective everything remains exactly as it was. It is especially powerful in light of the fact that the function prototypes may span multiple files, so a change to the function set could be put in a new header file and though massive changes may be made 'behind the scenes' wrt our STUDENT\_RECORD struct most of the project sees nothing happening.

=====

On to C++ and the change from code-centric to data-centric view.

In C++ the concept of the struct was not only expanded but also reinterpreted. Instead of a struct being a collection of data, in C++ a struct was reinterpreted to include within it the functions that modify, create, test and destroy it. The functions themselves are not members in the same sense that are the data members, however they are referred to as "member functions" of the struct. A new name was added "the class".

Re: Why "const rect& rhs" is used here?

## Re: Why "const rect& rhs" is used here?

There is no essential difference between "class" and "struct" in C++. They are one and the same except for the "public/private" default.

So, in C++ instead of saying this:

```
GetStudentName (&record, szNameOut, cntBuffSize);
```

we say this:

```
record.GetStudentName (char* szNameOut, cntBuffSize);
```

Where we define 'GetStudentName' \*inside\* the structure definition along with other members. Here are the two for comparison (I've dropped the word "Student" from the C++ version since it's functions ONLY refer to STUDENT\_RECORDs

```
// Regular struct
struct STUDENT_RECORD
{
char pcStudentFirstName[64];
char pcStudentLastName[64];

BOOL bHasTakenCourse[MAX_COURCES];
int iGrades[MAX_COURCES];

int iCurrGrade;
int iSkippedClasses;
BOOL bSomeBool;
// and so on.
};

// Funcs
void GetStudentName (const void* hcRecord, char* pcOut, size_t nBuffSize);
void SetStudentName (void* hRecord, const char* pcName);

//C++ struct
struct STUDENT_RECORD
{
private:
char pcStudentFirstName[64];
char pcStudentLastName[64];

BOOL bHasTakenCourse[MAX_COURCES];
int iGrades[MAX_COURCES];

int iCurrGrade;
int iSkippedClasses;
BOOL bSomeBool;
// and so on.
```

## Re: Why "const rect& rhs" is used here?

```
public:  
void GetName (char* szNameOut, cntBuffSizee) const;  
void SetName (const char* szNameOut);  
};
```

Notice that in `GetStudentName` we were able to tell the compiler that we did not intend to modify the structure `phRecord`. What about the C++ version? In the C++ version a `STUDENT_RECORD` pointer is in fact passed into `GetName`, but it does not appear in the argument list. So how can we tell the compiler that we don't intend to modify this structure? The answer is we put the `const` modifier following the function prototype. So:

```
void GetName (char* szNameOut, cntBuffSizee) const;
```

100% identically means this:

```
void GetName (const [secret STUDENT_RECORD* pRecord],  
char* szNameOut, cntBuffSizee);
```

just as in the old version (3 arguments).

What's more, `pRecord` actually does have a name. It's called "this". "this" is simply a `STUDENT_RECORD*` just like in the old days. I think the meaning should be fairly obvious.

Notice as well the public and private keywords. These keywords are used to limit access to the structure's elements thereby forcing the user to go through a function interface (if you so choose). There are advantages to this data-centric point of view and some disadvantages. For one thing, every time you want to add a member to your struct, you have to recompile the whole world again (unlike the `void*` solution). The advantages really don't start to become apparent until the concept of inheritance is introduced.

But that's another story...

– Alan Carre