# Modern C++ in embedded systems – Part 1: Myth and Reality

**Dominic Herity (/user/Dominic Herity)**

FEBRUARY 17, 2015



### Constructors and destructors

In C++, a constructor is a member function that is guaranteed to be called when an object is instantiated or created. This typically means that the compiler generates a constructor call at the point where the object is declared. Similarly, a destructor is guaranteed to be called when an object goes out of scope. So a constructor typically contains any initialization that an object needs and a destructor does any tidying up needed when an object is no longer needed.

The insertion of constructor and destructor calls by the compiler outside the control of the programmer is something that makes the C programmer uneasy at first. Indeed, programming practices to avoid excessive creation and destruction of so-called temporary objects are a preoccupation of C++ programmers in general. However, the guarantee that constructors and destructors provide - that objects are always initialized and are always tidied up - is generally worth the sacrifice. In C, where no such guarantees are provided, consequences include frequent initialization bugs and resource leakage.

### Namespaces

C++ namespaces allow the same name to be used in different contexts. The compiler adds the namespace to the definition and to name references at compile time. This means that names don't have to be unique in the application, just in the namespace in which they are declared. This means that we can use short, descriptive names for functions, global variables, classes, etc. without having to keep them unique in the entre application. **Listing 7** shows an example using the same function name in two namespaces.

```
// Namespace example
namespace n1 {
    void f() {
    }
    void g() {
        f(); // Calls n1::f() implicitly
    }
};
```

MOST READ

11.01.2013
Inline Code in C and C++ (/design/programming-languages-and-tools/4423679/Inline-Code-in-C-and-C-)

11.08.2013
How to know when to switch your SCM/version control system (/design/programming-languages-and-tools/4424039/How-to-know-when-to-switch-your-SCM-version-control-system)

10.26.2013
ARM design on the mbed Integrated Development Environment - Part 1: the basics (/design/programming-languages-and-tools/4423344/ARM-design-on-the-mbed-Integrated-Development-Environment---Part-1--the-basics)

EMBEDDED (/VIDEOS) TV (/VIDEOS) (/videos) video library (/videos)

```
namespace n2 {
    void f() {
    }
    void g() {
        f(); // Calls n2::f() implicitly
    }
};

int main() {
    n1::f();
    n2::f();
    return 0;
}
```

*Listing 7: Namespaces*

When large applications are written in C, which lacks namespaces, this is often achieved by adding prefixes to names to ensure uniqueness. See **Listing 8.**

```
/* C substitute for namespace */

void n1_f() {
}

void n1_g() {
    n1_f();
}

void n2_f() {
}

void n2_g() {
    n2_f();
}

int main() {
    n1_f();
    n2_f();
    return 0;
}
```

*Listing 8: C substitute for namespaces using prefixes*

### Inline functions

Inline functions are available in C99, but tend to be used more in C++ because they help achieve abstraction without a performance penalty.

Indiscriminate use of inline functions can lead to bloated code. Novice C++ programmers are often cautioned on this point, but appropriate use of inline functions can significantly improve both size and speed.

To estimate the code size impact of an inline function, estimate how many bytes of code it takes to implement it and compare that to the number of bytes needed to do the corresponding function call. Also consider that compiler optimization can tilt the balance dramatically in favor of the inline function. If you conduct actual comparisons studying generated code with optimization turned on, you may be surprised by how complex an inline function can profitably be. The breakeven point is often far beyond what can be expressed in a legible C macro.

### Operator overloading

A C++ compiler substitutes a function call when it encounters an overloaded operator in the source. Operators can be overloaded with member functions or with regular, global functions. So the expression `x+y` results in a call to `operator+(x, y)` or `x.operator+(y)` if one of these is declared. Operator overloading is a front end issue and can be viewed as a function call for the purposes of code generation.

### New and delete

In C++, `new` and `delete` do the same job as `malloc()` and `free()` in C, except that they add constructor and destructor calls, eliminating a source of bugs.

### Simplified container class

To illustrate the implementation of a class with the features we have discussed, let us consider an example of a simplified C++ class and its C alternative.

**Listing 9** shows a (not very useful) container class for integers featuring a constructor and destructor, operator overloading, new and delete. It makes a copy of an int array and provides access to array values using the `operator[]`, returning 0 for an out of bounds index. It uses the `(nothrow)` variant of new to make it easier to compare to the C alternative.

```
#include <iostream>
#include <new>

class int_container {
public:
```

```
        int_container(int const* data_in, unsigned len_in) {
            data = new(std::nothrow) int[len_in];
            len = data == 0? 0: len_in;
            for (unsigned n = 0; n < len; ++n)
                data[n] = data_in[n];
        }

        ~int_container() {
            delete [] data;
        }

        int operator[](int index) const {
            return index >= 0 && ((unsigned)index) < len?
data[index]: 0;
        }
    private:
        int* data;
        unsigned len;
    };

    int main() {
        int my_data[4] = {0, 1, 2, 3};
        int_container container(my_data, 4);
        std::cout << container[2] << "\n";
    }
```

*Listing 9: A simple integer container class featuring constructor, destructor, operator overloading, new and delete*

Listing 10 is a C substitute for the class in Listing 9. Operator overload `int_container::operator[](int)` is replaced with function `int_container_value(…)`. The constructor and destructor are replaced with `int_container_create(…)` and `int_container_destroy(…)`. These must be called by the user of the class, rather than calls being added automatically by the compiler.

```
    #include <stdio.h>
    #include <stdlib.h>

    struct int_container {
        int* data;
        unsigned len;
    };

    void int_container_create(struct int_container* this, int
const* data_in, unsigned len_in) {
        this->data = malloc(len_in * sizeof(int));
        this->len = this->data == 0? 0: len_in;
        for (unsigned n = 0; n < len_in; ++n)
            this->data[n] = data_in[n];
    }

    void int_container_destroy(struct int_container* this) {
        free(this->data);
    }

    int int_container_value(struct int_container const* this, int
index) {
        return index >= 0 && index < this->len? this->data[index]:
0;
    }

    int main() {
      int my_data[4] = {0, 1, 2, 3};
      struct int_container container;
      int_container_create(&container, my_data, 4);
      printf("%d\n", int_container_value(&container, 2));
      int_container_destroy(&container);
    }
```

*Listing 10: C substitute for simple string class*

Note how much easier to read `main()` is in Listing 9 than in Listing 10. It is also safer, more coherent, more maintainable, and just as fast. Consider which version of `main()` is more likely to contain bugs. Consider how much bigger the difference would be for a more realistic container class. This is why C++ and the object paradigm are safer than C and the procedural paradigm for partitioning applications.

All C++ features so far discussed confer substantial benefits at no runtime cost.

### Inheritance

In discussing how C++ implements inheritance, we will limit our discussion to the simple case of single, non-virtual inheritance. Multiple inheritance and virtual inheritance are more complex and their use is rare by comparison.

Let us consider the case where class B inherits from class A. (We can also say that B is derived from A or that A is a base class of B.)

We know from the previous discussion what the internal structure of an A is. But what is the internal structure of a B? We learn in object-oriented design (OOD) that

inheritance models an 'is a' relationship – that we should use inheritance when we can say that a B 'is a' A. So if we inherit Circle from Shape, we're probably on the right track, but if we inherit Shape from Color, there's something wrong.

What we don't usually learn in OOD is that the 'is a' relationship in C++ has a physical as well as a conceptual basis. In C++, an object of derived class B is made up of an object of base class A, with the member data of B tacked on at the end. The result is the same as if the B contains an A as its first member. So any pointer to a B is also a pointer to an A. Any member functions of class A called on an object of class B will work properly. When an object of class B is constructed, the class A constructor is called before the class B constructor and the reverse happens with destructors.

**Listing 11** shows an example of inheritance. Class B inherits from class A and adds the member `function B::g()` and the member `variable B::secondValue`.

```
// Simple example of inheritance

class A {
public:
    A();
    int f();
private:
    int value;
};

A::A() {
    value = 1;
}

int A::f() {
    return value;
}

class B: public A {
private:
    int secondValue;
public:
    B();
    int g();
};

B::B() {
    secondValue = 2;
}

int B::g() {
    return secondValue;
}

int main() {
    B b;
    b.f();
    b.g();
    return 0;
}
```

*Listing 11: Inheritance*

**Listing 12** shows how this would be achieved in C. `Struct B` contains a `struct A` as its first member, to which it adds a variable `secondValue`. The function `BConstructor(struct B*)` calls `AConstructor` to ensure initialization of its 'base class'. Where the function `main()` calls `b.f()` in Listing 11, `f_A(struct A*)` is called in Listing 12 with a cast.

```
/* C Substitute for inheritance */

struct A {
    int value;
};

void AConstructor(struct A* this) {
    this->value = 1;
}

int f_A(struct A* this) {
    return this->value;
}

struct B {
    struct A a;
    int secondValue;
};

void BConstructor(struct B* this) {
    AConstructor(&this->a);
    this->secondValue = 2;
}

int g_B(struct B* this) {
```

```
        return this->secondValue;
    }

    int main() {
        struct B b;
        BConstructor(&b);
        f_A ((struct A*)&b);
        g_B (&b);
        return 0;
    }
```

*Listing 12: C Substitute for inheritance*

It is startling to discover that the rather abstract concept of inheritance corresponds to such a straightforward mechanism. The result is that well-designed inheritance relationships have no runtime cost in terms of size or speed.

Inappropriate inheritance, however, can make objects larger than necessary. This can arise in class hierarchies, where a typical class has several layers of base class, each with its own member variables, possibly with redundant information.



(mailto:?subject=Modern C++ in embedded systems – Part 1: Myth and Reality&body=http://www.embedded.com/design/programming-languages-and-tools/4438660/2/Modern-C--in-embedded-systems---Part-1--Myth-and-Reality)

< Previous (/design/programming-languages-and-tools/4438660/1/Modern-C--in-embedded-systems---Part-1--Myth-and-Reality) Page 2 of 3 Next > (/design/programming-languages-and-tools/4438660/3/Modern-C--in-embedded-systems---Part-1--Myth-and-Reality)

## 16 COMMENTS

WRITE A COMMENT

**Freddie Chopin (/User/Freddie%20Chopin)** **POSTED: DEC 3, 2016 5:12 PM EST**

Yeah... This whole mythology surrounding C++ in embedded will probably never change. But I'm also trying to fight this myth by doing something that many people will consider a complete nonsense - I'm writing an RTOS for ARM Cortex-M microcontrollers in C++11 - http://distortos.org/ (http://distortos.org/) . So far it's working perfectly fine, and now I don't have to write all those C++ wrappers for "other" RTOSes to use a member function in a thread or to have a mutex globaly initialized by a constructor. Not to mention all the simplification caused by RAII and similar patterns not even imaginable in pure C. Try using a lambda with 10 arguments as a thread function or passing a complex object via message queue. Not possible in a typical RTOS written in C. Quite easy to do in C++11.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**Dominic Herity (/User/Dominic%20Herity)** **POSTED: DEC 18, 2016 2:00 PM EST**

Writing a small footprint RTOS in C++11 is a good way to demonstrate its suitability for the environment.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**Freddie Chopin (/User/Freddie%20Chopin)** **POSTED: JAN 25, 2017 3:35 AM EST**

Hello Dominic and thanks for your reply!

Personally I find the importance of size to be a bit overstated. Don't get me wrong - I don't say that it's OK for the RTOS to use 1MB of flash and 256kB of RAM just to blink a LED - I'm far from that. But at the same time I also far from saying that any scheduler which uses more than 1000 bytes of flash and 50 bytes of RAM is bloated and should never be used. The numbers are obviously picked at random, but I guess you see my point. It's all about balance and trade-offs.

The first reason is that the size of firmware is usually directly proportional to the features it offers. A framework which provides 10 "features" will be smaller than a framework which provides the same 10 "features" and one feature more. A RTOS which provides detachable threads or POSIX signals will be bigger than a RTOS which only provides the bare minimum.

Reason number two - the size and speed are often in opposition. Bubble sort will no doubt be smaller than heapsort, but the later will definitely be faster. Usually (not always, but quite often) the same is true for any kind of algorithm or piece of code.

Last but not the least - for large projects using C++ is quite a good idea, as it simplifies a lot of things. Projects which need multithreading are usually large anyway. In a large project - which uses ~100kB of flash - it makes little difference whether a RTOS uses 1kB or 10kB of flash, especially when some parts of these 10kB are most likely used by the application anyway (some standard functions from C or C++ libraries).

Anyway - the whole RTOS that I've written fits into a typical ARM Cortex-M microcontrollers without problems, leaving plenty of space for the application (;



**OdinHolmes (/User/OdinHolmes) POSTED: JUL 4, 2016 6:07 AM EDT**

I think a lot of C fans are missing the point. If you write C++ code like you would write C code there is no difference. If you write C++ code like your were writing code for a desktop then it will be slow and huge. C++ brings new tools and new tools bring new paradigms and patterns. Most notably zero cost abstraction brings encapsulation of expertise. Templates bring -smarter- libraries, policy based class design and static aliasing. Its a brave new world and we need more library devs ;)



**Dominic Herity (/User/Dominic%20Herity) POSTED: DEC 18, 2016 2:02 PM EST**

All true.



**FredHansen (/User/FredHansen) POSTED: APR 24, 2016 6:55 PM EDT**

It's not quite true that C++ is a superset of C. Many of the things are violate the superset idea are good (stricter type checking for example).

I agree that C++ does not always result in larger compiled code. However, final implementation are often bigger and slower when written in C++. I think the issue is that slick/cute new options in C++ (new/delete, templates, multiple inheritance, ...) are like new toys for FW engineers. We can help playing with them. The final product is often bigger as a result.

I was writing embedded system code back when everyone said you had to use assembly and C was the new comer. We worried that the final code would be too big and too slow, ... Eventually compilers got good enough and fast processors and memory got cheap enough that C became the right answer. Also the problems got complex enough that we just could not reasonably attack them with assembly.

The same thing will likely happen again. Either C++ or some other OO language will eventually dominate embedded systems. If there is that much C code around then we simply not there yet.

I'd also point out that C++ is a LOT bigger than C. It has a lot of toys in it. The transition might happen sooner if C++ were a bit smaller.



**Dominic Herity (/User/Dominic%20Herity) POSTED: APR 25, 2016 2:11 PM EDT**

All true. My motivation for the original 1998 article was to challenge the often stated proposition that C++ is _inherently_ unsuitable for embedded systems. It was aimed at C programmers who were thinking about trying it, but deterred by inaccurate assertions.
I wanted to shed some light on how it works and suggested that it be adopted piecemeal to maintain productivity and minimize risk while getting comfortable with the language. I hope it convinced some people to try it and that they found the experience rewarding.



**HannaD573 (/User/HannaD573) POSTED: APR 14, 2016 10:35 AM EDT**

Why I will never use c++ for small mcus:
1. Too often, it is assumed that a highly skilled C engineer can just start using C++ as a better "C". Not true - C++ has a learning curve far beyond "C". I dare say many C++ engineers do not fully understand C++.

2. When I malloc, I know the exact size of the memory block, and need to carfully track and control this - when I "new" an object I really have no idea how much I malloc'ed - further the object could possibly have contained objects that are allocated in the ctor.

3. Due to the constrained size of the environment, I can't really use OO design effectively. Instead of your Quixote crusade why not accept that this is not going to happen, and instead push for better embedded design.

Clean Code is not about the language.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**Dominic Herity (/User/Dominic%20Herity) POSTED: APR 15, 2016 5:04 PM EDT**

I don't just assume that C++ can be used as a better C. I assert it.
Take your example of malloc versus new. I don't agree that malloc if preferable, but there's nothing to stop you using it in C++, if that's what you really want.
Anything you can do in C, you can do in C++. No problem.
My own experience is that OOD makes code clearer when it reaches somewhere between 100 and 1000 lines, so I would contend that OOD can improve clarity even in systems with a few KB of code.
Of course, C++ offers more than just OOD - exceptions, templates, etc. Templates can benefit very small systems by doing compile time calculations that can't be done in C.
Of course, small systems have to be handled with skill and are unforgiving of poor design choices.
Think of C++ as a bigger toolbox than C. Some of the extra tools should be used with care, but don't blame the toolbox if you misuse a tool. The purpose of the article was to help C programmers understand how some of the extra tools work, so that they can make an informed decisions about when to use them.
I'm all for better embedded design and refusing to use a more powerful, more expressive, language because it can be misused is a mistake. All languages can be misused and more powerful ones can be misused more.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**Wouter Van Ooijen (/User/Wouter%20van%20ooijen) POSTED: JAN 3, 2016 11:56 AM EST**

I'd like to add one disadvantage of virtual functions: they are an inbline & optimization barrier. This can make a hughe difference in performance (in addition to code size) for functions at or near the edges of the call tree. when the object relations are known only at run time, this is unavoidable. But for situations where the object tree is known at compile time compile-time composition via templates can (IMO nicely) solve this problem. (As I argue in "Objects? No Thanks", check http://www.embedded.com/design/programming-languages-and-tools/4428377/Objects--No--thanks---Using-C--effectively-on-small-systems-(http://www.embedded.com/design/programming-languages-and-tools/4428377/Objects--No--thanks---Using-C--effectively-on-small-systems-) or https://www.youtube.com/watch?v=k8sRQMx2qUw (https://www.youtube.com/watch?v=k8sRQMx2qUw) )

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**Dominic Herity (/User/Dominic%20Herity) POSTED: JAN 3, 2016 3:07 PM EST**

This is true. Thanks for pointing it out. I was comparing a virtual function to a function in a separate translation unit that the compiler can't access for optimisation, but this can't be assumed.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**OdinHolmes (/User/OdinHolmes) POSTED: JUL 4, 2016 6:13 AM EDT**

Building on the idea in "Objects? No Thanks" policy based class design in general allows "open closed" idiom with zero overhead. A well designed C++ library can have a fraction of the flash footprint of its C counterpart.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**Onkar Raut (/User/Onkar%20Raut) POSTED: JUN 12, 2015 1:20 PM EDT**

Out of curiosity, what would be your opinion on working with peripheral functions? From my experience, singleton classes are a fair choice, but seem to be an inappropriate when you have multiple iterations of the same peripheral e.g. a timer function.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**Dominic Herity (/User/Dominic%20Herity) POSTED: JUN 17, 2015 12:21 PM EDT**

Singletons have many problems. See http://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons. (http://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons.)
For cases like you describe, I tend to use a function that returns a reference to an object that is statically allocated inside the function.
Such an object is constructed the first time the function is called, which gives you lazy initialization, one of the good things about the singleton.
So you can do something like this:

```
class uart;
uart& console() {
static uart the_console(...);
return the_console;
}
```

Its a little more complicated if multiple threads can make the first call.

If you have multiple objects of the same class, the function could take a numeric parameter and return a reference to the correct one. This case is more complex because the objects will generally need different constructor parameters, so a statically allocated array won't do it.
Something like:

```
class timer;
timer& system_timer(unsigned n) {
static std::vector<timer> the_timers;
if (the_timers.empty()) {
// Set them up
}
return the_timers[n];
}
```

You could also use a template like this.

```
template <unsigned n> system_timer() {
static timer the_timer(...); // constructor oparameters depend on n.
return the_timer;
}
```

You'd need to make sure the template is only instantiated once for each n.
This has the benefit that you get a compile time error if you try to access a non-existent device.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**OdinHolmes (/User/OdinHolmes)** **POSTED: JUL 4, 2016 6:31 AM EDT**

Here my rebuttal of the SO post for embedded specific:
[1] embedded programs are smaller which mitigates the problem, model the singleton as a template class and you can always pass the class object around (its empty anyway the optimizer will remove it in most cases) as a kind of handle.

[2] its usually most efficient to initialize at start up any way in embedded so initialization is less of an issue. There are cases where initialization order is an issue which is really the last standing argument against singletons.

[3] if your singletons are template classes you can use the traits pattern to mock them out, no show stopper here.

[4] mock them out in unit tests. Endless object lifetime avoids the heap (fragmentation, non deterministic timing etc.) and is therefore an advantage in my mind. Heap must be as big as the worst case set of simultaneously used objects plus heap overhead plus fragmentation penalty. If the singleton can be part of the worst case set of objects removing it from the heap and giving it static lifetime is actually a huge RAM savings (saves overhead and fragmentation penalty) although this may seem counter intuitive at first.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**

**OdinHolmes (/User/OdinHolmes)** **POSTED: JUL 4, 2016 6:15 AM EDT**

Make the peripheral instance a template parameter, its like function overloading but for singletons.

**REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F2%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)**