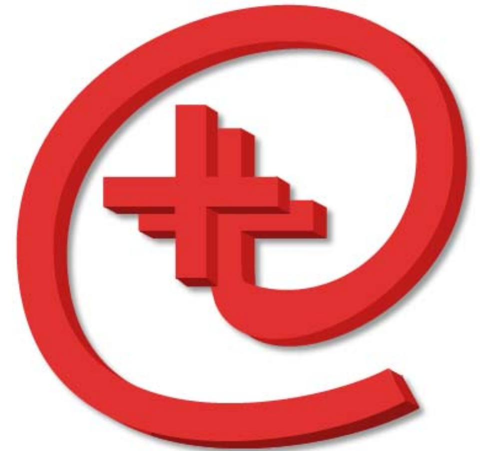
BECK THE POINT OF SYNERGY

Application note: Programming C++ on the IPC@CHIP®

Introduction



This application note discusses the points which should be considered by the programmer before developing C++ applications for the IPC@CHIP®. There are no restrictions concerning the C++ programming language on the IPC@CHIP®. Although there are some points which should be noted for programming smart and memory saving software. Furthermore there are some hints how to get access to the operating system and the hardware of the IPC@CHIP® since this could be more difficult when using C++. The following descriptions apply to both IPC@CHIP® development environments (Paradigm C++ and Borland C++).

All chapters are marked with a symbol above the chapter title. The symbol describes the type hint(s) in the chapter. The table below lists type of hints with their corresponding symbol and meaning.

● Memory	Marks a chapter which contains hints describing how to create memory saving programs
● Functionality	Marks a chapter which contains hints on technical or functional themes.
● Real-time	Marks a chapter which contains hints concerning the theme "real-time"
● Tip	Marks a chapter which contains general programming tips

Have also a look to the C++ example programs available on our website (http://www.beck-ipc.com/ipc).

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany
E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

● Tip

How to create a C++ project

Create a project as described in the IPC@CHIP® *Getting Started* document which is available on our website.

The files which contains C++ code must be named *.cpp (e.g. myapp.cpp). So give your C++ files the extension *CPP* and add them to the project. Figure 1 shows an example project window.

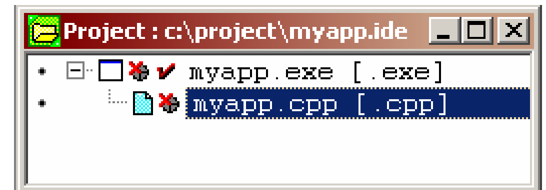


Figure 1

● Functionality

Class methods as callback

Problem:

Some @CHIP-RTOS functionalities expect a function pointer, for example the task-functions-pointer, CGI-function-pointer, several callback function like socket-callback a.s.o.

Sadly the C++ standard does not support pointers to class methods (except static class methods; they will be handled like functions).

Example:

In figure 2 the *main* function assigns the pointer of the function named *myFunction* to the variable *fp*. The next statement of *main* tries to assign the pointer of the method named *myMethod* to the variable *mp*. This is not allowed so the compiler will give an error.

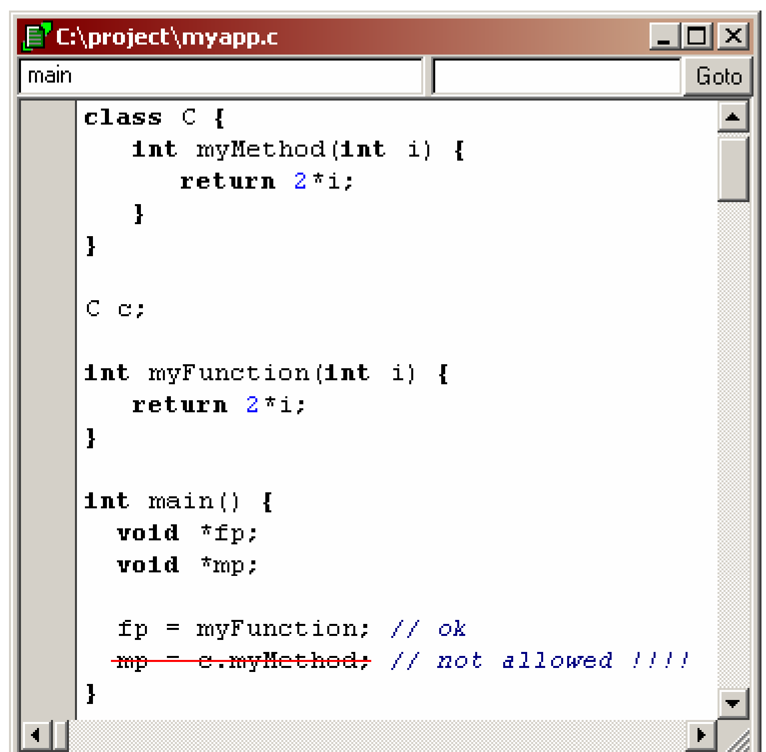


Figure 2

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany
E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

Conclusion:
A solution to this problem is creating a wrapper function. The pointer to the wrapper function will be given to the @CHIP-RTOS. In the wrapper function the class method will be called.

Example:
Figure 3 shows a simple example program with the implementation of the wrapper function named *theWrapperFuntion*. In *main* the pointer of the wrapper function will be given to the @CHIP-RTOS (e.g. as task function, CGI function, socket callback, etc.) When the wrapper function is called, it calls the method *myMethod* of the instance *c* of the class *C.*

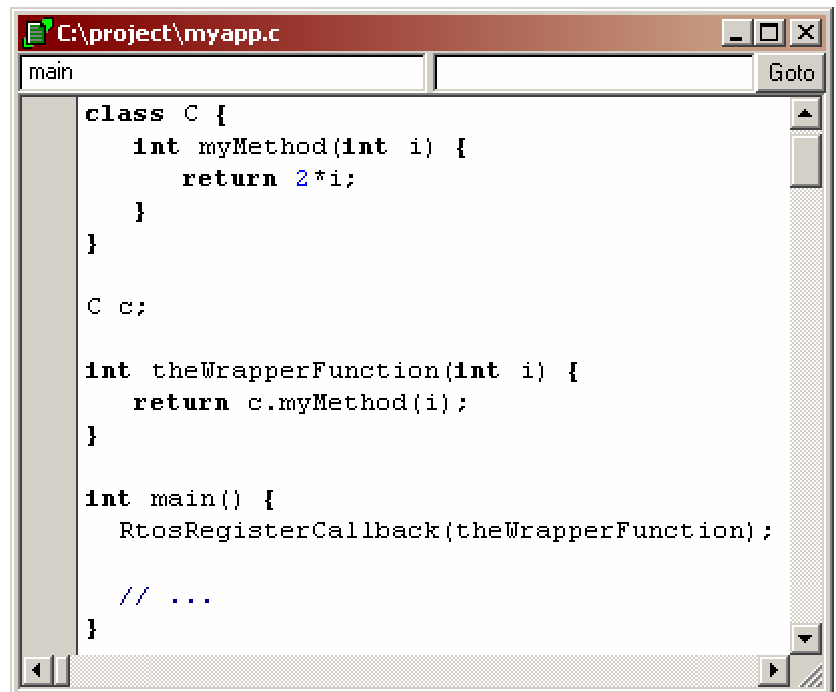


Figure 3

To be more flexible you could create a global list, in which the wrapper function searches the according object and then calls its method. For an implementation example to that theme have a look into the C++ examples available on our website.

Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany

E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

Calling conventions for functions and class methods

Problem:
The calling conventions define the name, the structure of the stack frame and the responsibility for the cleaning up the stack. It is helpful to know something about calling conventions if you intend to call C++ functions/methods from assembler or if you deal copiously with callback functions and function pointers.
The C++ calling conventions are mostly identical with the C calling conventions. Except for three points, the C++ calling conventions are identical to the C calling conventions.

Conclusion:
So if you want to use a C++ symbol in assembler you have to note the points below.

1) The generation of the assembler name:
In C the assembler name will be generated by prefixing a underscore character to the C name.
Example:

```
myFunction(int i);    -> _myFunction
```

In C++ the function name will be generated by integrating the parameter types into the name.
Example:

```
myFunction(int i);    -> @myFunction$qi
```

For class methods, the class name will be integrated into the assembler name as well.
Example:

```
C :: myMethod(int i); -> @C@myMethod@qi
```


© Beck IPC GmbH

Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany

E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

2) The calling of non-static class methods:
Non-static class methods expect an instance pointer to their object instance as an implicit parameter on the stack. The caller has to put the Segment:Offset pointer as last parameter pushed onto the stack (segment pushed before offset).

3) Function and Methods with default parameter:
The caller of the function or method with default parameters has to take care that all parameters are supplied on the stack. (Also the default parameter(s)!)

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany
E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

Using a struct or a class as return value of a function or a class method

In C++ functions and class methods could return complex data types (that means a structure instance or a class instance).

1) Structure instance

Problem:
A function (or class method) which returns a structure instance expects a pointer to such an instance as parameter on the function call. So the caller (the calling code) creates an instance of the structure on the stack before. Then it calls the function and assigns a pointer to the created instance as a parameter. Note that the caller creates for every function call a single instance on the stack. So if you call *n* times the same function the caller creates *n* instances.
If the return value of the function call will be assigned to some variable (another structure instance), the content of the stack instance will be copied into the variable.

Example:
Figure 4 shows a simple example program. The *main* function calls the function *f* two times. The red marked variables in *main* (*tmp_s1, tmp_*s2) are allocated implicitly by the compiled code. So every call of *f* costs 4 bytes stack memory (one instance of *S* requires 4 bytes).

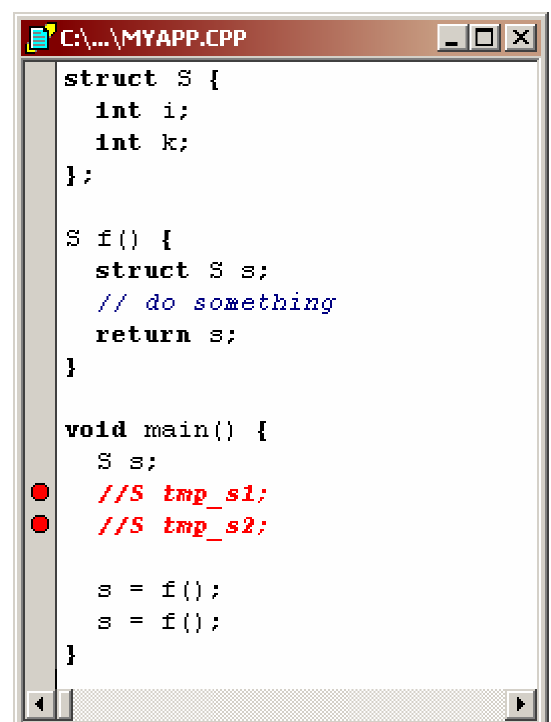


Figure 4

Conclusion:

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany
E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

Pass structures by reference parameter instead of as return value. This will save some stack memory, because it eliminates the stack instance and also it saves the copy operation.

Example:
Figure 5 shows the modification of the preceding program. Here the instance of the structure *S* will not be used as a return value anymore. The function *f* gets the instance of *S* as reference parameter.

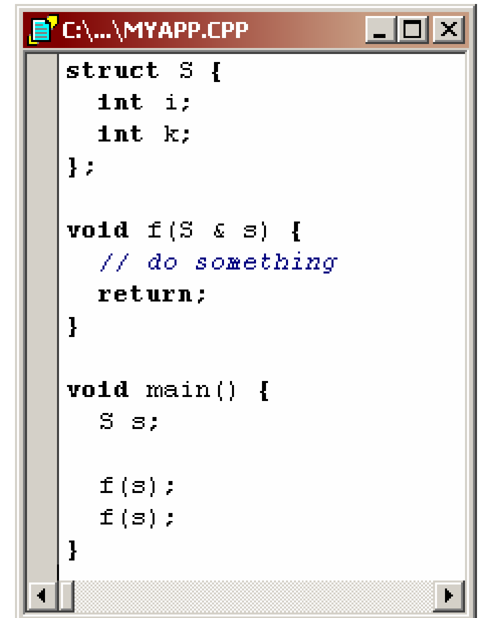


Figure 5

2) Class instance

Problem:
A function which returns a class instance expects also a pointer to such an instance. But the caller does not create an instance for every function call. It passes a pointer to the variable to which will be assigned the return value from the function. This saves the stack instance and also the copy operation for the assignment.
Only if there is no assignment there will be created dummy instances on the stack and the procedure described above (see *Structure instance*) will be used.

Conclusion:
A better way is to pass class instances by reference parameter instead of a as return value. This can save some stack memory, because it saves the stack instance in the case that the function will be called without assignment of the return value.

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany

E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

Local Variables

Problem:
The C standard specifies that local variables have to be declared at the beginning of a function. In contrast the C++ standard allows local variables to be declared inside the code where ever you want.
But regardless of the location in the code where the local variable is declared, the variable is allocated on the stack for the entire execution time of the function (from the functions start until its end). That means only the scope of the variable is bounded - not its durability. Important to note is that the compiler could not detect the life span of variables in different sections. So it allocates additional stack memory for each section's variables.

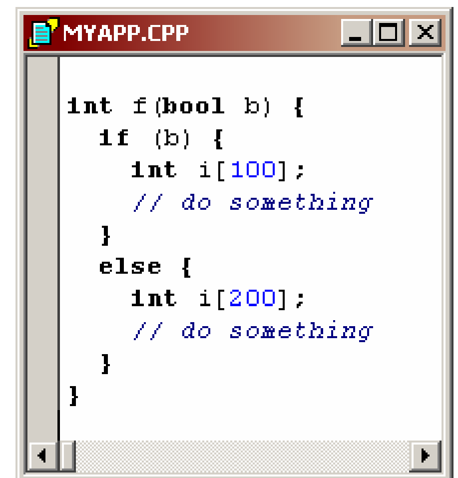


Figure 6

Example:
The example function *f* in figure 6 requires about 600 bytes stack memory (300 integers). This is because the compiler allocates both **int** arrays separately on the stack.

Conclusion
Declare your variables at the beginning of the function. This way you are sure how much stack memory the function requires.

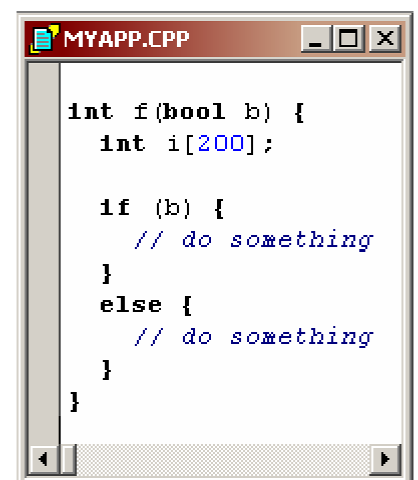


Figure 7

Example:
The function *f* in figure 7 requires about 400 bytes stack memory (200 integers). Both sections of the if-statement use the same variable.

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany

E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

Non-constant initializers

The C++ standard allows non-constant expressions (such as functions) for the initialization of local variables as well as the initialization of global variables. The time when the initialization will be performed depends on the kind of variable being initialized (global, local or static-local).

1) Global variables (as well as static class attributes)

Problem:
A global variable will be initialized by the startup code of the application. So the non-constant initialization expression will be evaluated before the *main* function will be called. That means that all functions which initialize global variables will be called before main. It is important to consider that at this time all other global variables may not be initialized! So be careful using global variables in non-constant initializers or its functions.

Example:
Figure 8 shows a program which generates a *DIVIDE BY ZERO* exception without any code in the main function. This is because at time of initialization of *g1000Ticks* the Variable *gOneAdd* is not yet initialized.

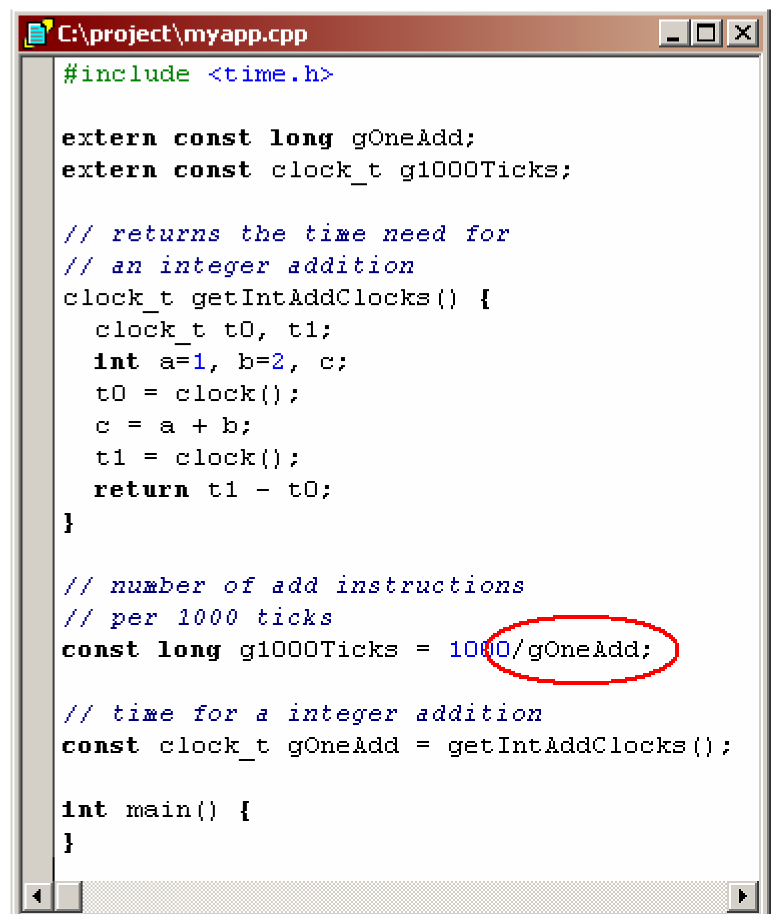
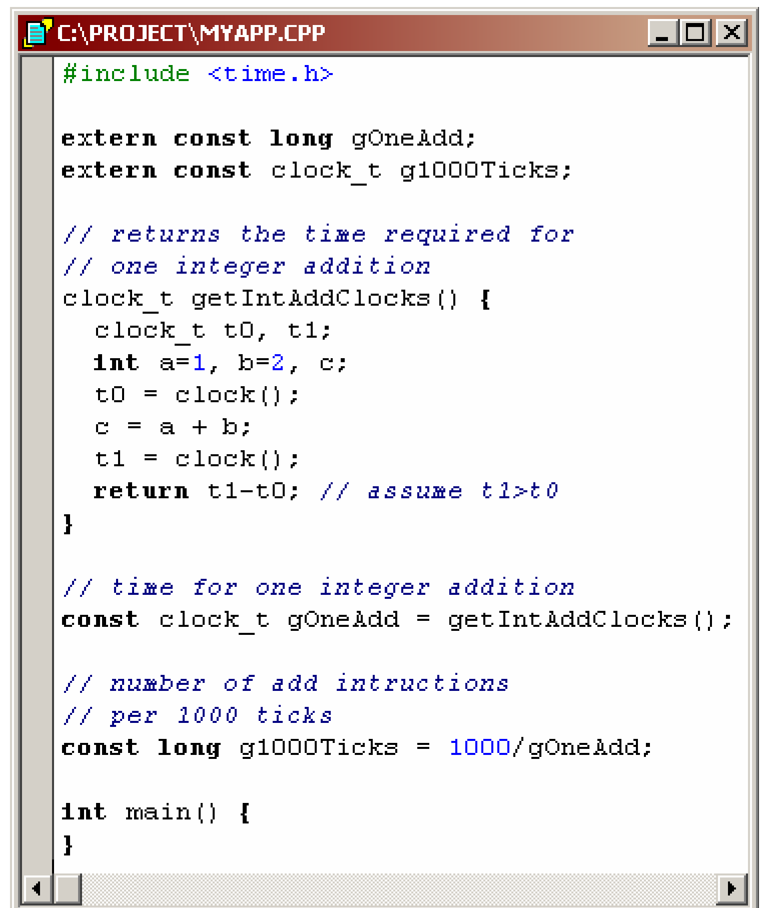


Figure 8

Conclusion:
Be careful using non-constant initializers. Make sure that all global variables used by your non-constant initializer are initialized before use. This includes all global variables which are used by functions called by your non-constant initializers. The initialization order for global variables is the order of the declaration of the variables. That means that the initialization of a variable declared at the beginning of a file will be done first.

Example:
In figure 9 your can see almost the same program as in figure 8. Only the declaration order of the two global variables is changed. So now at the initialization time of *g1000Ticks*, the variable *gOneAdd* has already been initialized.




```
#include <time.h>

extern const long gOneAdd;
extern const clock_t g1000Ticks;

// returns the time required for
// one integer addition
clock_t getIntAddClocks() {
  clock_t t0, t1;
  int a=1, b=2, c;
  t0 = clock();
  c = a + b;
  t1 = clock();
  return t1-t0; // assume t1>t0
}

// time for one integer addition
const clock_t gOneAdd = getIntAddClocks();

// number of add intructions
// per 1000 ticks
const long g1000Ticks = 1000/gOneAdd;

int main() {
}
```

Figure 9

Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany

E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

2) Non-static (normal) local variables

Normal local variables will be initialized at the beginning of each function call (before any function code is executed). Note that local variables must not be initialized by the function itself or functions which call this function. Otherwise an endless loop results.

Example:

Figure 10 shows an example program. The initialization of the local variable *i* is not allowed. A call of function *f* results in an endless loop. Note that in that case the compiler does not generate an error!

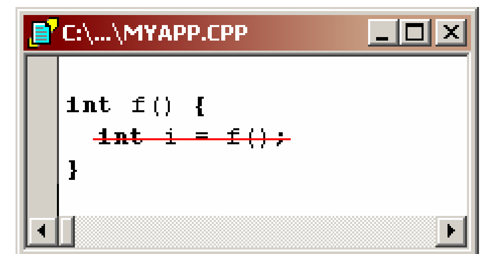


Figure 10

3) Static local variables

Static local variables will be initialized only on the first function call. So the execution time of the first function call will be longer than the subsequent calls (due to the time required for evaluation of the non-constant initializer(s)). Also there is again the potential for creating endless loops as mentioned above for non-static local variables.

© Beck IPC GmbH

Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany

E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

Exception handling

The C++ standard supports a comfortable exception handling. It is available with the *try*, *catch* and *throw* keywords. For building programs which should use C++ exceptions you have to assure that exception handling is enabled for your project. Therefore you have to check two options.

The first option is found in the target expert. The target expert will be opened when creating a new project. If you have already created a project you can reopen the target expert with a right mouse click on the *.exe* node in the project window. In the target expert window the option *No Exceptions* has to be disabled in order to use the C++ exception handling. Figure 11 shows the target expert with the necessary disabled option.

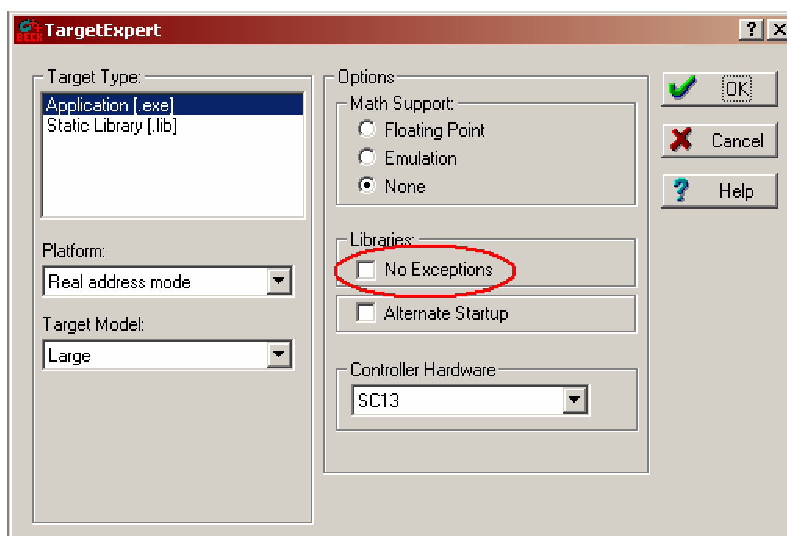


Figure 11

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany

E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

The second option you can find in the project options of your project. To get there, click on the *Options -> Project* menu in your development environment. In the project options windows you can see several notes. Click on the note *C++ Options -> Exception Handling/RTTI*. Now you can see some check boxes on the right side of the window. The option *Enable exceptions* has to be checked when using C++ exception handling. Figure 12 illustrates the project option window and the option which must be enabled.

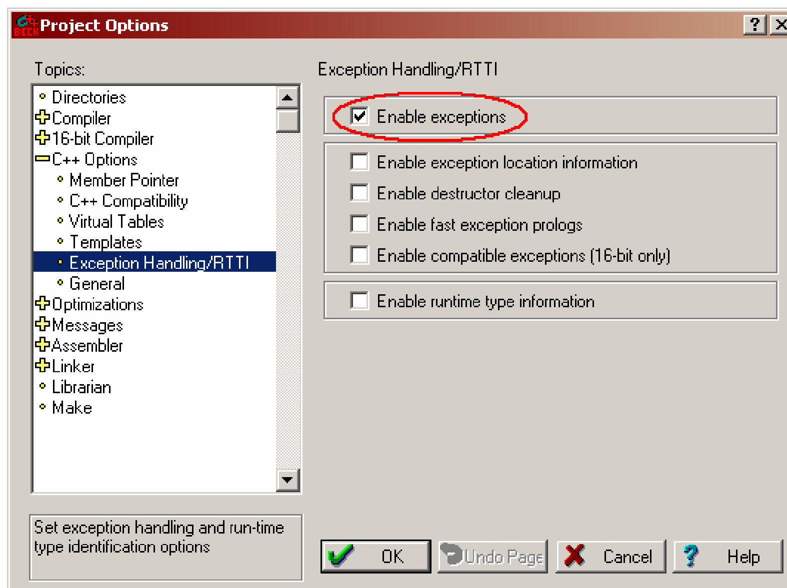


Figure 12

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany
E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

C++ class library

Both IPC@CHIP® development environments (*Paradigm* and *Borland*) includes some C++ classes called *class library*. The *class library* supports typical functionalities for object-oriented software development like streams, strings, etc. A description of the hole *C++ class library* is available in the development environments help system. Figure 13 shows an abstract of the class library description. In *Paradigm* you can get there by clicking on the *Help* -> *Content* menu and opening the *Paradigm C++ Class Libraries Reference Guide* book in the *Content* tab. If you use the *Borland IDE* open the *classlib.hlp* help file in the *Help* directory which you can find in the Borland installation path on your hard disc (typical "C:\Program Files\Borland\BC5\Help").

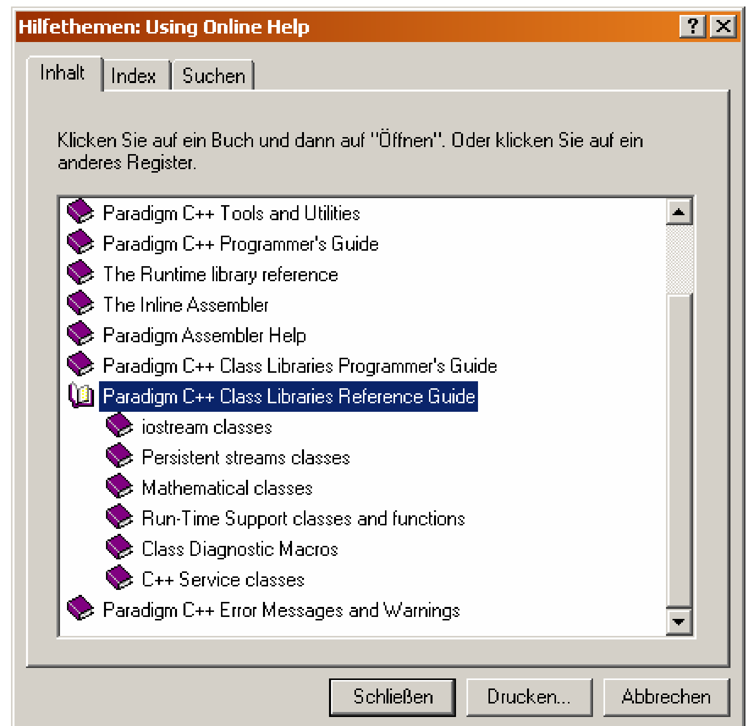


Figure 13

Note that not all classes described in the help system are supported by the IPC@CHIP® controller. The table below shows which classes could be used for IPC@CHIP® applications:

iostream classes:	full supported
Persistent streams classes:	not supported
Mathematical classes:	full supported
Run-Time Support classes and functions:	full supported
Class Diagnostic Macros:	not supported
C++ Service classes:	the classes *string*, *TSubString* and *TRegexp* are supported only.

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany
E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group

Summary

Below you can see a short summary of all important points discussed in this document.

1) Non static class methods cannot be referenced by function pointers.
2) Assembler names of C++ symbols are named differently than in C.
3) Non-static class methods expect an instance pointer as an implicit parameter.
4) If a function is defined using a default parameter, the default parameter has to be passed by the caller.
5) Using class instances or structure instances as return value of functions or class methods could costs some stack memory.
6) Declaring local variables inside the code costs stack memory over the whole lifetime of the function.
7) If a function contains local static variables, the first function call takes longer than the subsequent function calls (because of the initialization).
8) Initializing a local variable by using the function itself or another function which calls that function will result in an endless loop.
9) When using global variables in initializers of other global variables, be sure that all data used by the initializer are already initialized.
10) For using C++ exception handling, the exception handling must be enabled for the project.

© Beck IPC GmbH
Beck IPC GmbH
Grüninger Weg 24
35415 Pohlheim/Germany
E-Mail: info@beck-ipc.com
www.beck-ipc.com
A member of the Festo group