To:
Tenouk

C LAB WORKSHEET 3
Building and Running An Empty Win32 Console Application Project 1

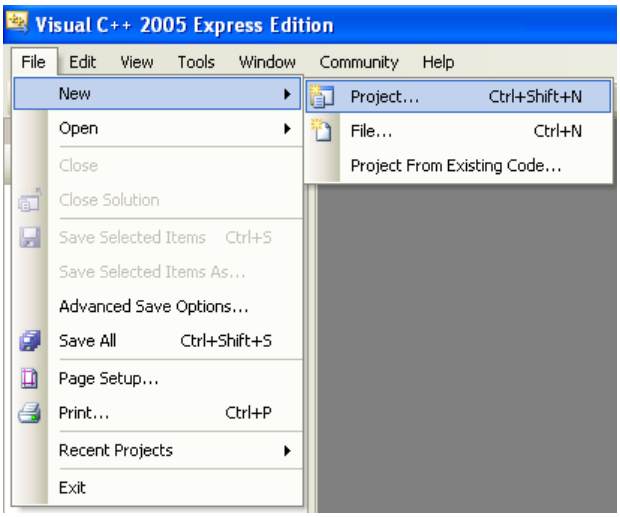
In this page, we will be getting familiar with compiler. The related tutorial references are: C/C++ intro & brief history, C/C++ data type 1, C/C++ data type 2, C/C++ data type 3 and C/C++ statement, expression & operator 1, C/C++ statement, expression & operator 2 and C/C++ statement, expression & operator 2.

How To Build And Run Your First An Empty Win32 Console Application Project

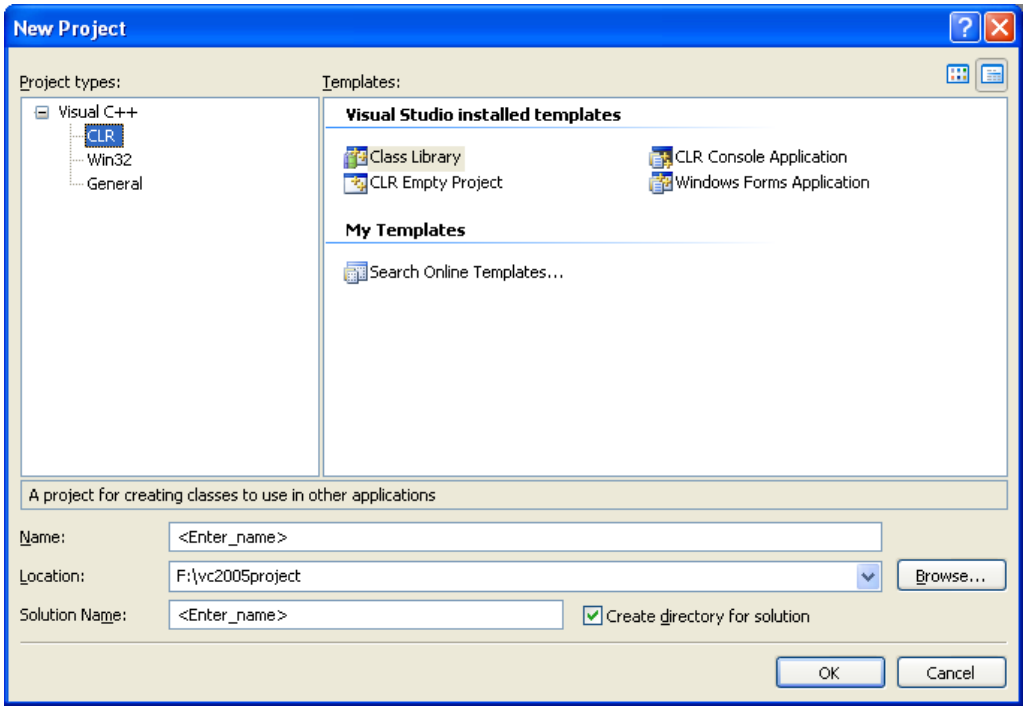
The OS used is Windows XP Pro with Service Pack 2 and compiler is Visual C++ 2005 Express Edition (VC++ 2005 EE).

Now we are going to create, build and run a very simple C program. During the process we will explore some of the common Visual C++ 2005 EE functionalities. You should be familiar with the steps in this module because the same steps will be used for all the lab practice.

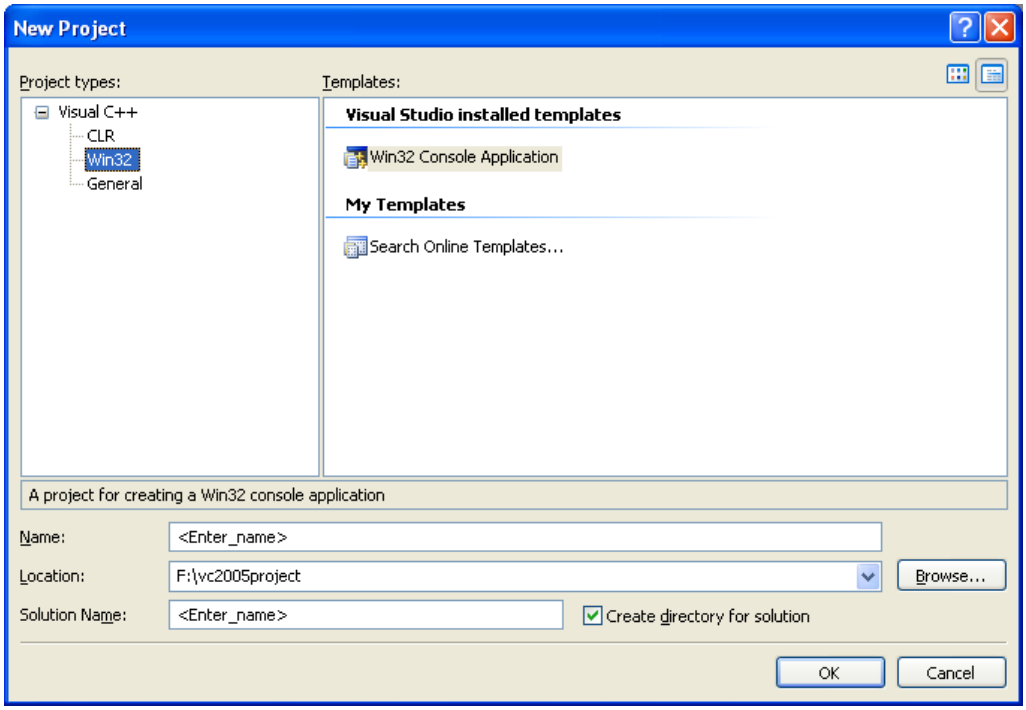
1. First of all launch your VC ++ 2005 EE IDE. Click the **File ⟶ New ⟶ Project** menu.



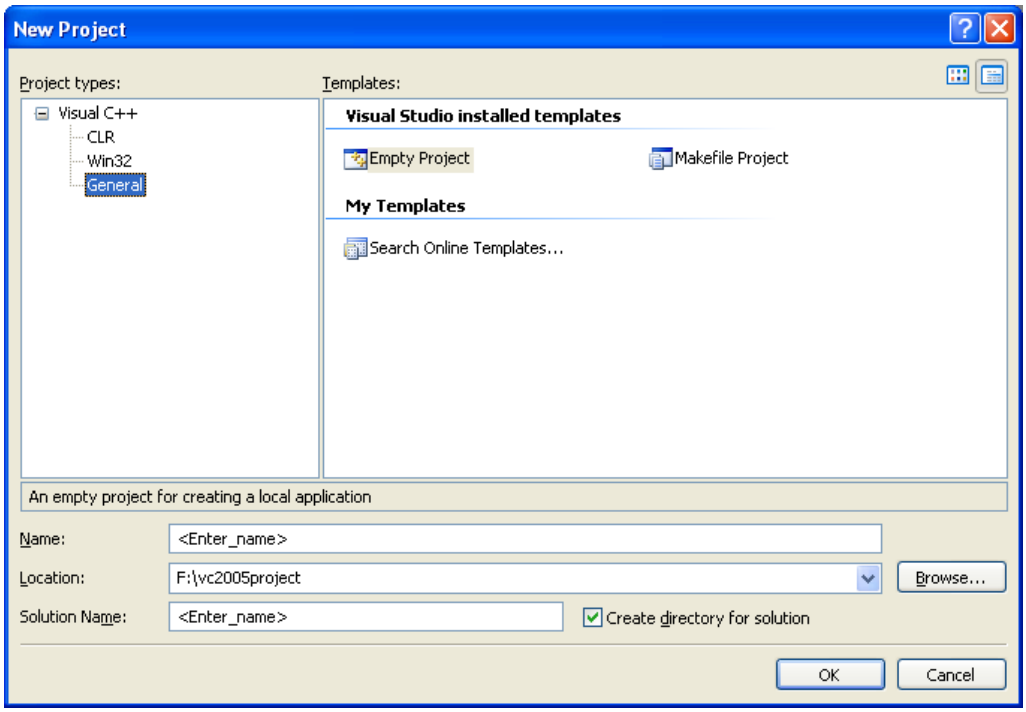
2. The **New Project** form will be launched. On the left window we have **CLR**, **Win32** and **General** project types. On the right we have templates for those project types respectively.



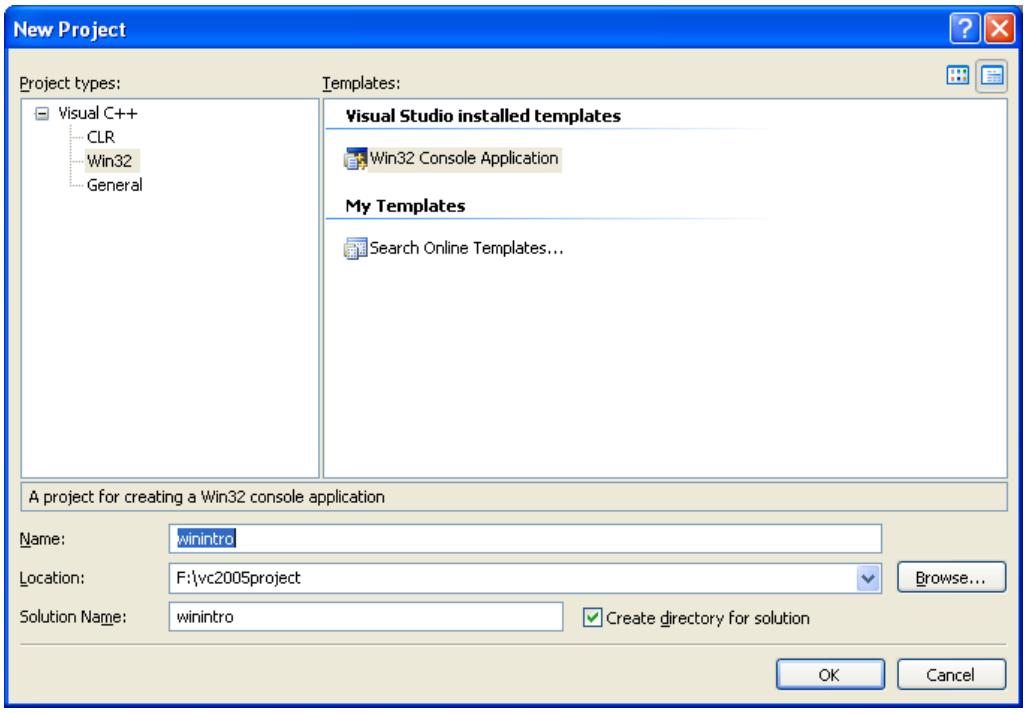
3. The **Win32** project template is **Win32 Console Application**.



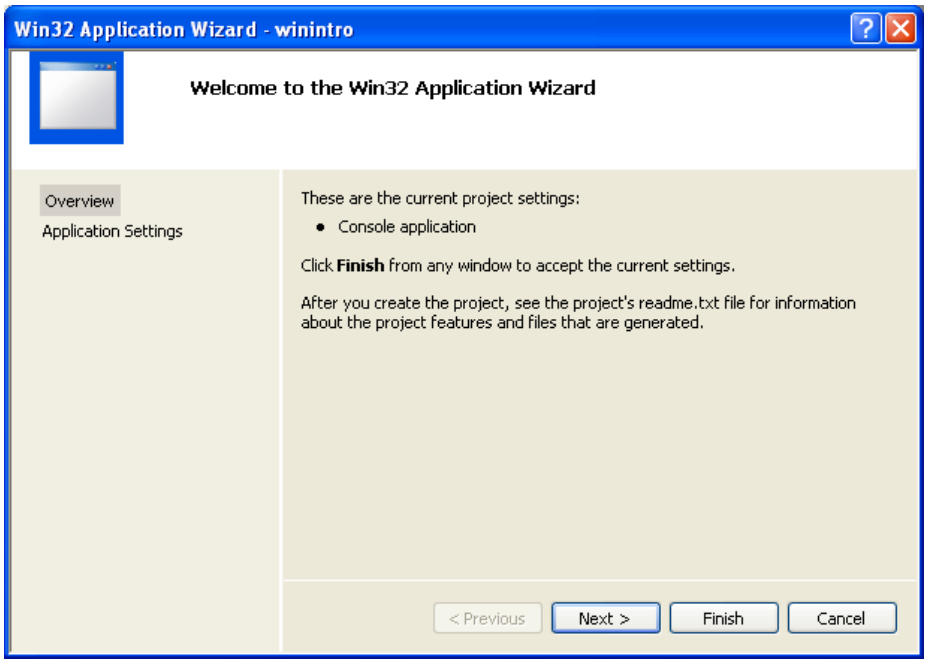
4. The **General** project types contain an **Empty** and **Makefile** projects.



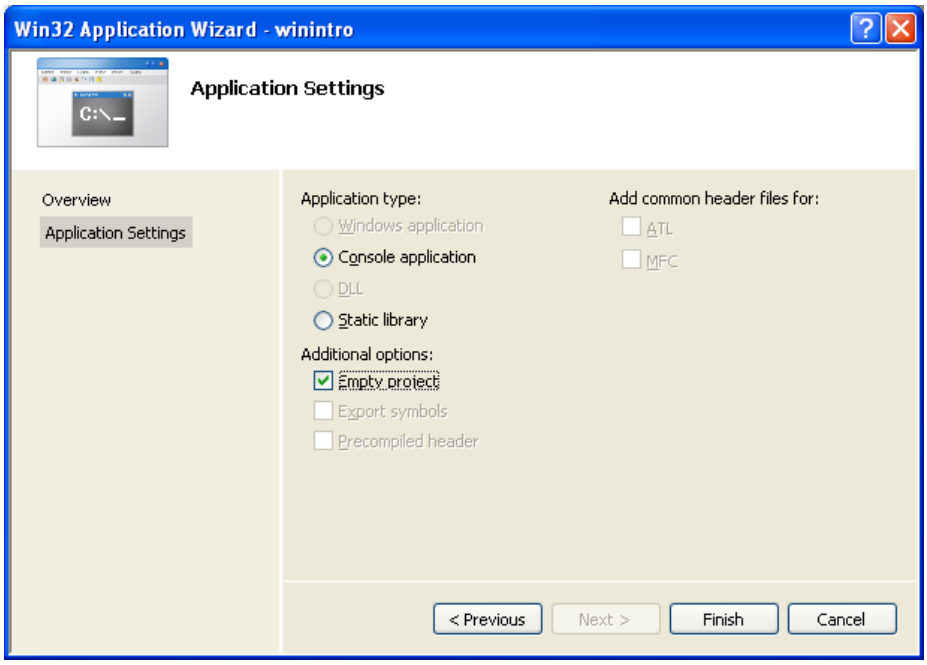
5. Select the **Win32** project type and **Win32 Console Application** template as shown below. Put the project name in the **Name:** field. Change the project (and all it files) **Location:** as needed. The solution default name is similar to the project name, change it if required and then click the **OK** button.



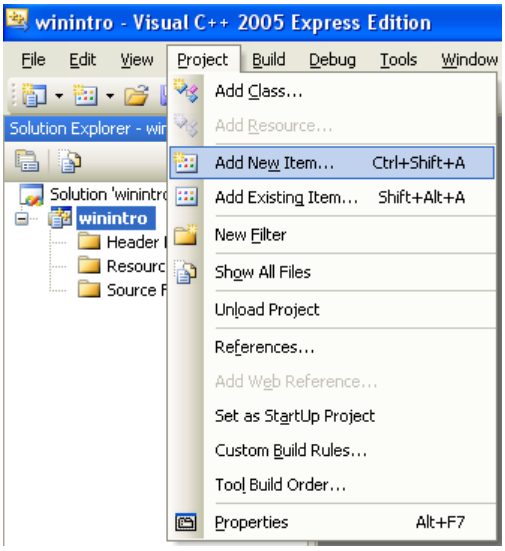
6. **Win32 Application Wizard** form will be launched. In this form we will further refine our Win32 console application settings. The **Overview** page gives a summary about our project.



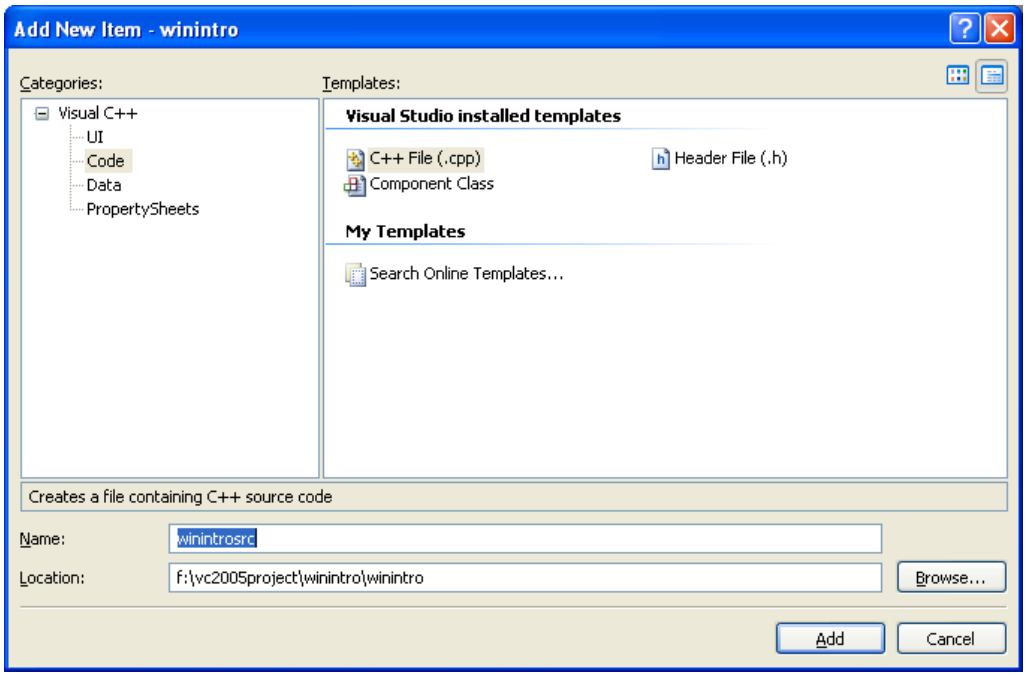
7. Select the **Application Settings** on the left. Select **Console Application** under **Application Types** and **Empty Project** under **Additional Options**. Then click the **Finish** button.



8. Our project should be launched in VC++ 2005 EE. Now we only have an empty Win32 console application project. We need to add file to the project. As a beginning let add C source file to the project. Select **Project** → **Add New Item** menu.



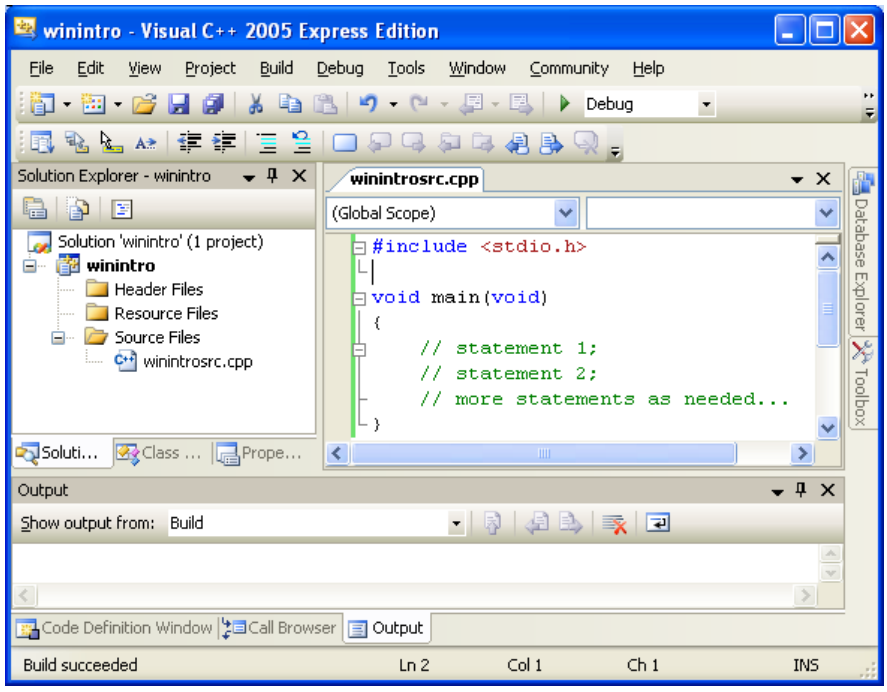
9. The **Add New Item** form will be displayed as shown below. We have several categories of project items and their respective templates as well. For this exercise select **Code** under **Categories** and **C++ File (.cpp)** under templates. Put the source file name in the **Name:** field. The **Location:** should be default to our project. Then click the **Add** button.



10. Next, just type the following loose C sample code in the editor (right window).

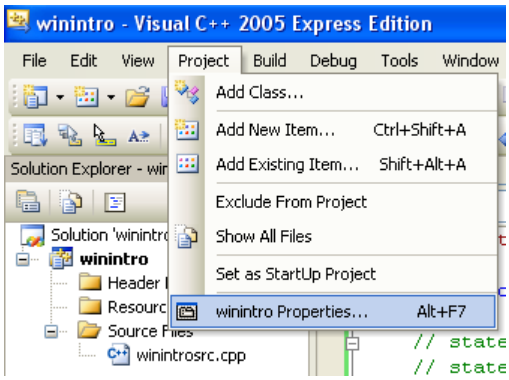
```
#include <stdio.h>

void main(void)
{
    // statement 1;
    // statement 2;
    // more statements
     // as needed...
}
```

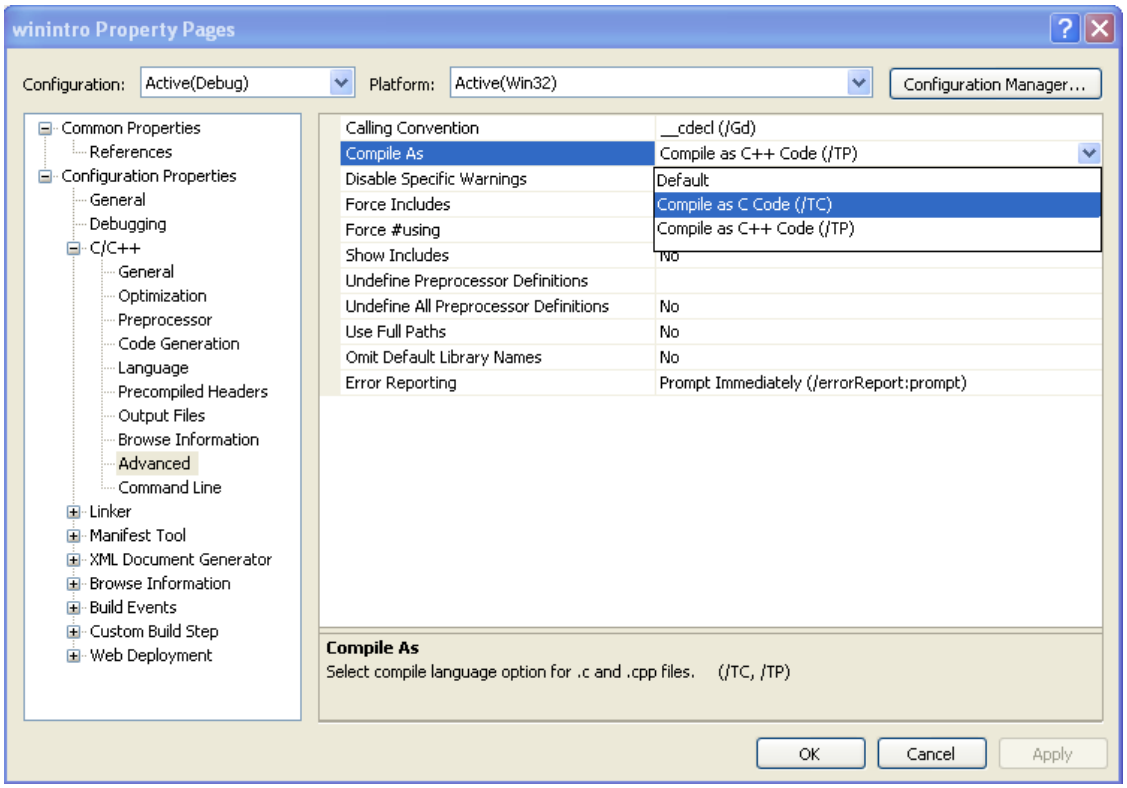


C, C++ and Project Properties Story

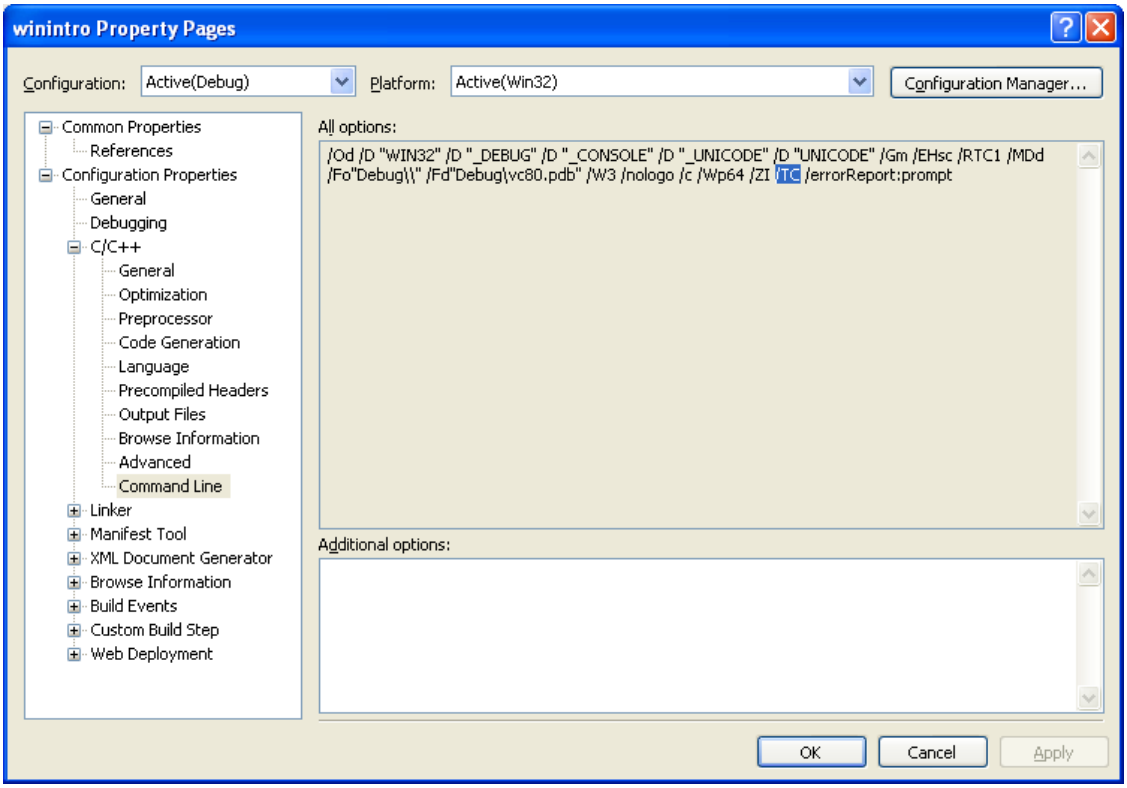
To compile the C codes, give your files the .c extension (if created using text editor and compiled using command line), for example **mysourcecode.c**. The Visual C++ compiler automatically assumes that files with the .c extension are C files and not C++ files, and rejects C++ syntax and keywords (such as public, private, and class). C++ files use the .cpp extension. For the command line compilation, **cl.exe** (CL) is a tool that controls the Microsoft C and C++ compilers and linker. cl.exe can only be run on Windows 2000, Windows XP and Windows Server 2003 operating systems. Similarly, by default, CL assumes that files with the .c extension are C source files and files with the .cpp or the .cxx extension are C++ source files. Another way (as used in this module and that follows) you can use the project property page to set this setting as shown below. Select **Project** ⟶ **your_*project_name*** **Properties** menu.



1. All the project settings can be changed/enabled/disabled here. Expand the **Configuration Properties** folder and then expand the **C/C++** subfolder. Click the **Advanced** link. Click the **Compile as C++ Code (/TP)** of the **Compile As** (default). From the list select **Compile as C Code (/TC)**.



2. Click the **Apply** button and then select **Command Line** link on the left windows. Notice the **/TC** option, means compile as C codes.



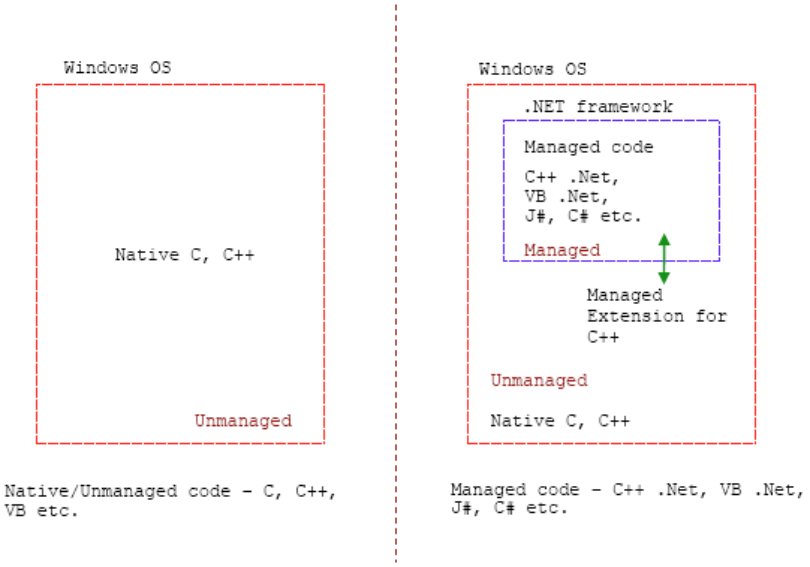
The /Tc option specifies that filename is a C source file, even if it does not have a .c extension. The /Tp option specifies that filename is a C++ source file, even if it doesn't have a **.cpp** or **.cxx** extension. A space between the option and filename is optional. Each option specifies one file; to specify additional files, repeat the option.
/TC and /TP are global variants of /Tc and /Tp. They specify to the compiler to treat all files named on the command line as C source files (/TC) or C++ source files (/TP), without regard to location on the command line in relation to the option. These global options can be overridden on a single file by means of /Tc or /Tp as shown below.

/Tcfilename
/Tpfilename
/TC
/TP

The Managed, Unmanaged and Managed Extensions for C++ Code Story

A **managed code** is an application program (such as C# program) that is executed within a runtime engine (such as .Net framework and Java Virtual Machine (JVM)) installed in the same machine. The application cannot run without it. The runtime environment provides the general library of software routines that the program uses and typically performs memory management and etc. It may also provide just-in-time (JIT) conversion from source code to executable code or from an intermediate language to executable code. Java's JVM and .NET's Common Language Runtime (CLR) are examples of runtime engines. In simple words a manage code means the code can be managed in the aspects such as type safety and memory management.
The **unmanaged code** is an executable program that runs by itself, launched from the operating system, the program calls upon and uses the software routines in the operating system, but does not require another software system to be used. C/C++ programs compiled into machine language for a particular platform and Assembly language programs that have been assembled into machine language are examples of unmanaged code.
Before this we just have C/C++ code that is unmanaged. Managed Extensions for C++ (or just Managed C++) are extensions to the Visual C++ compiler and language to allow them to create .NET code and enable access to the functionality of the .NET Framework. They include a set of **Keywords** and **Attributes** to extend the C++ language to work with, and generate, managed code. There are also some additional **Pragmas**, **Pre-processor Directives**, and **Options** for the compiler, as well as some **Linker** options. If you notice the Managed Extension of C++ uses C++ keywords and syntax, but they follow .NET rules for types and facilities.
Managed Extensions for C++ is Microsoft's set of deviations from C++, including grammatical and syntactic extensions, keywords and attributes, to bring the C++ syntax and language to the .NET Framework. Managed C++ is not a complete standalone, or fully fledged programming language. These extensions allow C++ code to be targeted to the Common Language Runtime (CLR) in the form of managed code as well as continue to interoperate with native code (unmanaged).

For specific definition, "Managed" refers to that it is run in, or managed by, the .NET virtual machine that functions as a sandbox for enhanced security in the form of more runtime checks, such as buffer overrun checks. Additionally, applications written in Managed C++ compile to MSIL - Microsoft Intermediate Language - and not directly to native CPU instructions like regular C++ applications do.
Managed C++ code can interoperate with any other (managed) language that also targeted for the CLR such as C# and Visual Basic .NET as well as make use of features provided by the CLR such as garbage collection. This means Managed C++ occupies a unique position in the gallery of .NET languages. It is the only language that can communicate directly with .NET languages (such as C#, VB.NET) and native C++. The other .NET languages can only communicate with C++ code via PInvoke or COM, rather slow and inefficient methods. So we have C++ code (unmanaged), Managed C++ (managed) and C++ .Net (and other .Net languages such as VB .Net and C# -managed). But since Managed C++ can communicate directly in both managed and unmanaged contexts, it is often used as a "bridge". This story can be described in the following Figure.

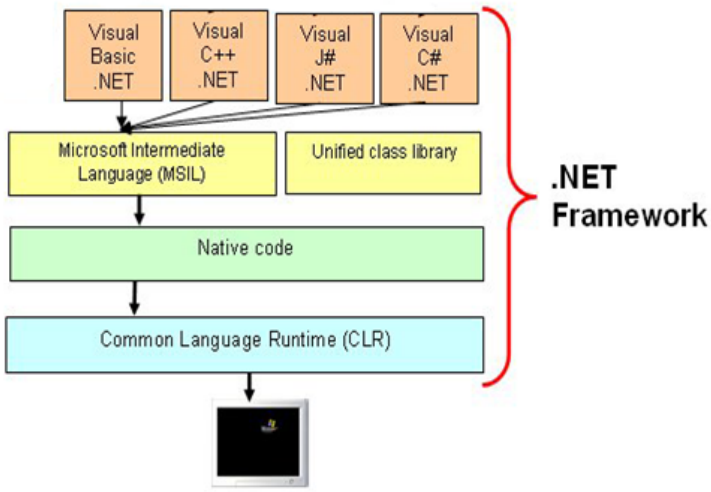


C++ And Common Language Infrastructure (CLI)

The Managed C++ extensions were significantly revised to clarify and simplify syntax and expand functionality to include managed generics. These new extensions were designated C++/CLI and included in Microsoft Visual Studio 2005. C++/CLI (Common Language Infrastructure) is the newer language specification that superseded Managed Extensions for C++. Completely reviewed to simplify the older Managed C++ syntax, it provides much more clarity over code readability than Managed C++. Like Microsoft C# (ECMA-334), C++/CLI is standardized by European Computer Manufacturers Association (ECMA-372) and ISO/IEC . It is currently only available on Visual C++ 2005.
Common Language Runtime (CLR) is the name chosen by Microsoft for the virtual machine component of their .NET initiative. It is Microsoft's implementation of the Common Language Infrastructure (CLI) standard, which defines an execution environment for program code. The CLR runs a form of bytecode called the Common Intermediate Language (CIL, Microsoft version is MSIL). The CLR runs on Microsoft Windows operating systems.

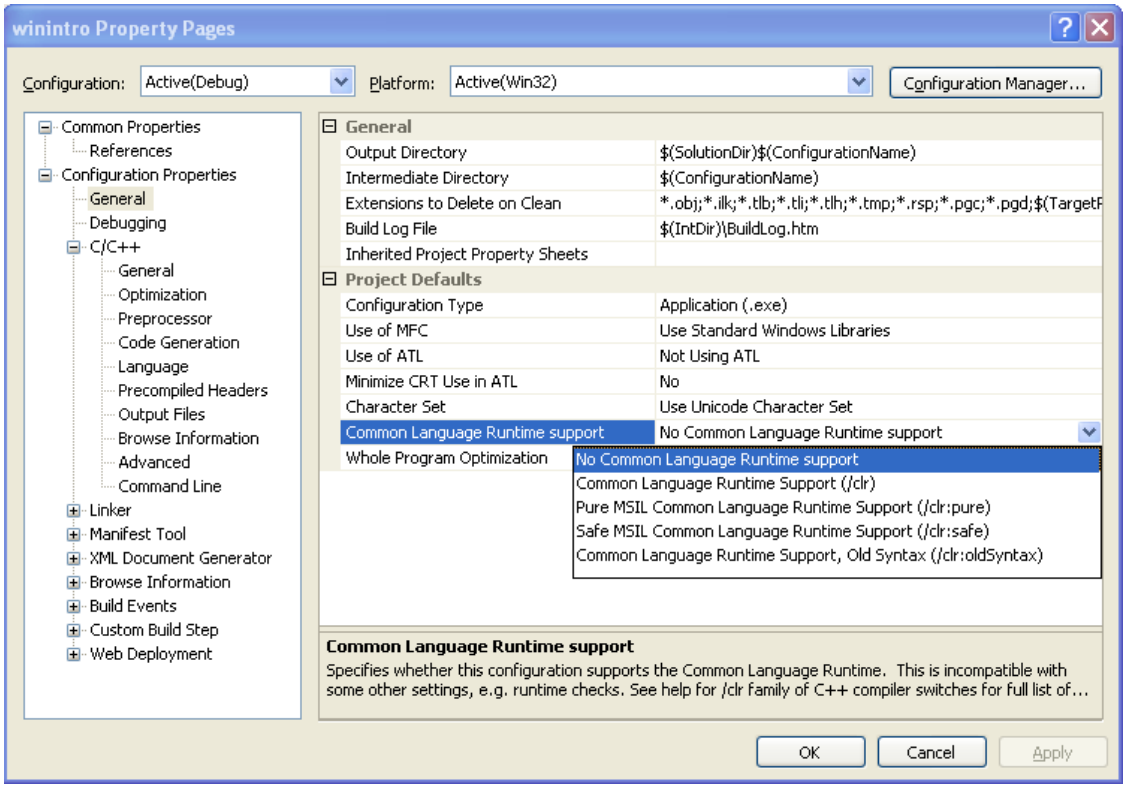
Machine Symbolic Intermediate Language (MSIL)/Common Intermediate Language (CIL)

In general, **Common Intermediate Language** (CIL) is the lowest-level human-readable programming language in the **Common Language Infrastructure** and in the .NET Framework. Languages which target the .NET Framework for example, compile to CIL, which is assembled into **bytecode**. CIL resembles an object oriented assembly language, and is entirely stack-based. It is executed by a virtual machine. The primary .NET languages are C#, Visual Basic .NET and Managed C++. CIL was originally known as **MicroSoft Intermediate Language** (MSIL). Due to standardization of C# and the Common Language Infrastructure, the bytecode is now officially known as CIL. CIL is still often referred to as MSIL, however, and has been backronymed to mean **Machine Symbolic Intermediate Language**. This is especially true of longtime veterans of the .NET languages.
Microsoft Intermediate Language (MSIL) is a byte-code that Microsoft .NET technology uses to accomplish platform independence and runtime safety seems similar to already matured Java. During compilation of .NET programming languages, the source code is translated into MSIL code rather than machine-specific object code as done previously in C/C++.



MSIL is a CPU and platform independent instruction set that can be executed in any environment supporting the .NET framework. MSIL code is verified for safety during runtime, providing better security and reliability than natively compiled binaries.

See all the Hello World example in various programming languages at Wiki that include the CLI.
Click the **General** link under the **Configuration Properties** folder. By default /clr compiler option (Common Language Runtime support) is not enabled as shown below.



In VC++ 2005 EE, functions are managed by default when /clr compiler option is used. The compiler ignores the managed and unmanaged pragmas if /clr is not used in the compilation.
Each implementation of C and C++ supports some features unique to its host machine or operating system. Some programs, for instance, need to exercise precise control over the memory areas where data is placed or to control the way certain functions receive parameters. The #pragma directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages. Pragmas are machine- or operating system-specific by definition, and are usually different for every compiler. Pragmas can be used in conditional statements, to provide new preprocessor functionality, or to provide implementation-defined information to the compiler.
For .NET Programming, Visual C++ supports the creation of three distinct types of components and applications (this is more visible if you compile through the command line):

clr option	Description
mixed (compiled with **/clr**)	Contains both unmanaged and managed parts, making it possible for them to use .NET features, but still contain unmanaged code.
pure - (compiled with **/clr:pure**)	Can contain both native and managed data types, but only managed functions.
verifiable - (compiled with **/clr:safe**)	Generates verifiable assemblies, like those written in Visual Basic and C#, conforming to requirements that allow the common language runtime (CLR) to guarantee that the code does not violate current security settings.

Table 1

All three are available through the /clr (Common Language Runtime Compilation) compiler

option. Fortunately, we are not going to use the CLR functionalities in this course. All program examples are unmanaged Win32 Console Mode Application.

Build, Run And Debug C Program: Part 1 | Part 2