Brian C. Becker's Compiler Tutorial

Your How-To Guide on Creating Real-World Compilers

Contents at a Glance:

Table of Contents	iii
About the Author	iv
Acknowledgements	v
Chapter 1 – Introduction	1
Chapter 2 – Gearing Up	8
Appendix A – Installing Dev-C++	23

Table of Contents

Cover..... i
Contents at a Glance:..... ii
Table of Contents..... iii
About the Author..... iv
Acknowledgements v
Chapter 1..... 1
Overview..... 1
Intended Audience..... 2
Structure..... 3
.NET Support..... 3
Hardware Requirements..... 3
Operating System Requirements..... 4
Software Requirements..... 4
Notations Used..... 4
Symbols Used..... 5
My Website:..... 6
Conclusion..... 6
Chapter 2..... 7
Introducing KoolB..... 7
Simply KoolB..... 8
Compilers vs. Interpreters..... 9
The Compile Process..... 11
What Languages?..... 12
Making a Tool's Inventory..... 14
A C++ Compiler..... 15
Creating Space for KoolB..... 16
Choosing an Assembler..... 16
Choosing a Linker..... 17
Choosing a Resource Compiler..... 18
Conclusion..... 18
Installing Dev-C++..... 21

About the Author

Brian C. Becker first become interested in programming during his high school years. He dabbled in C and some freeware programming languages he found on the Internet. After learning many variants of BASIC, he went on to learn C and C++, which soon became his language of choice. Assembly language, PHP, Perl, C#, and many other languages soon followed.

He became interested in compilers when his favorite programming language RapidQ went commercial. Although it took several years, he was determined to build his own compiler. During the hard process, he decided to document his progress so that others desiring to build their own compilers could have an easy to read tutorial to follow without having to learn everything in the school of hard knocks.

Brian is currently attending UCF (University of Central Florida) in Orlando for a BS in Computer Engineering. Involved in the UCF Robotics club, the Intelligent Systems Lab, and his church youth group, he leads an active life outside of academics. Currently, his attentions are turning more towards artificial intelligence and how new techniques can be applied to the ever expanding world of programming.

Acknowledgements

Thanks to Many People:

- First and foremost, I thank my Lord God Jesus for his unfailing love and patience when I mess up.
- A close second, I want to thank my family, who puts up with me always being on the computer working on something and their interest in my projects.
- My sister, who so kindly points out when my designs aren't good and helps re-design them.
- Steve of QDepartment group (http://groups.yahoo.com/group/QDepartment), who has put many hours to make the group a great resource for beginner compiler developers like myself. I owe him much for his patient answers to my sometimes stupid questions.
- Jared, who has stuck with compilers and showed great interest in my tutorial.
- Slowbyte, for suggesting a better memory management scheme for my compilers
- Ryan for reminding me to work on the Linux version, and for his helpful comments.
- Everybody at RQCompiler group, who have often given input and suggestions.
- Jack Crenshaw, who first started me writing compilers with his "Let's Build A Compiler" tutorial series.
- William Yu, who first got me interested in programming with his Rapid-Q compiler.
- Jeremy Gordon, author of GoLink and the NASM development team.
- Many others who I cannot thank enough.

Thanks Everybody!

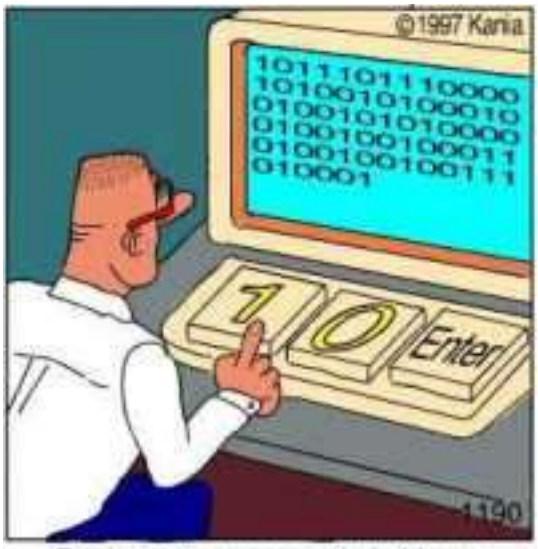
Chapter 1

Introduction

Welcome to my tutorial!

Here you will begin to explore what I call compiler technology: the ins and outs of building compilers. You will learn from the ground up how to design a programming language and then create a compiler.

I will show you how to create a simple, yet powerful BASIC programming language and compiler using C++. At times, building compilers can be frustrating, but often it is fun and rewarding. This is especially true when you consider that compilers saved us from what this cartoon calls real programming.



Real programmers code in binary.

Fortunately for us, we don't have to be 'real' programmers – we have 'real' compilers! All humor aside, I hope you find this tutorial easy to understand and fun to read. Enjoy!

Overview

Developing compilers can be a daunting task. I'm here to make that job easier by providing you with a 'step-by-step' handbook showing you exactly how you can build a compiler using C++. I will show you how to start off with a simple C++ project and develop it into a full-blown real-life compiler.

The compiler you will learn to build will be a BASIC compiler. Why BASIC? BASIC has long been considered an ideal language for teaching because of its simplicity, so I will continue that tradition. Also, BASIC code compiles quite easily, making your job less difficult. Once done, the compiler will be able to create simple, yet complete programs for Windows or Linux. In addition, I hope you will branch out and add personal touches to the language as I show you how to design and build it.

You will notice that I try to simplify things as much as possible. When two routes appear, I typically choose the easier. You might see my choice as less efficient or slower, but I aim to teach, not develop commercial or efficient compilers. And if I sling phrases like "Well, if you want to implement a recursive descent parsing engine with a STL enhanced tokenizing capabilities and an infinite number of look-a-heads, with profiling support to determine the efficiency coefficient of highly optimized inline routines…" I think you agree that it just doesn't work too well, to put it mildly. That would put even me to sleep.

Intended Audience

I do assume some things about you. First, I assume you are a driven genius overachiever who handles several jobs in addition to a full load of academic courses for your 5th doctorate. You stay up until midnight studying followed by several hours of relaxing where you might program 2048 bit fractal encryption programs. You only sleep when forced to.

Ok, perhaps I assume a bit too much!

However, as a writer, I aim for particular readers – a group of people who will benefit from what I write. I call this my intended audience.

- You are an average programmer and somehow compilers have caught your attention, so you are reading this to find out more. If this is you, welcome aboard and I think you will find compilers a fascinating subject as you learn to design and build them.

- You have thought of a great programming language or want to develop a compiler, but might not know where to start. Great! You've come to exactly the right place. I'll teach you how to develop a compiler and discuss important issues that relate, no matter what language you've thought up.
- You are a student and need that extra boost to understand compilers or complete that project. In this case, you may or may not have come to the right spot. Some courses use 'compiler compilers,' something I won't cover. If your professor uses these, you had best find a newsgroup or a tutorial that details their use. Otherwise, come on in and join the learning.

This tutorial does deal with some advanced computer related technologies – you just cannot escape that. I'll do my best to minimize the learning curve, but to avoid teaching you everything; I do assume some things about you:

- You can use Windows or Linux. If I have interrupted your party celebrating your mastery of double-clicking, this tutorial is probably not for you!
- You can program decently in C or C++. If you don't know C++ very well, but can pick it up as we go along, that's fine too. You can check out some resources on the Internet: www.cplusplus.com and www.cprogramming.com.
- You understand the basic concepts of assembly language. Ouch! This might not hit home for you, but don't worry. I will explain all assembly language I use as thoroughly as I can. Paul Carter has written a great e-book on modern PC assembly language, and I highly recommend you check it out. You can find his easy-to-read e-book at www.drpaulcarter.com/pcasm.

Structure

I have structured this tutorial like a how-to guide: each chapter builds on previous ones. I begin by laying the foundation for the compiler. You might find this a little uninteresting to begin with, but a good foundation alleviates much pain later on. I will then move onto designing the BASIC programming language and showing you how to develop the compiler to produce working programs. Lastly, I will show you some advanced topics concerning the compiler, such as optimization.

.NET Support

A promising new technology is emerging from Microsoft called the .NET Framework. Although it resembles a rip-off of Java, it goes much deeper. I won't go into much detail other than I am closely watching the development of .NET as it influences future compiler developments. At this point, I don't see a need to abandon traditional programming methods, but in the future, I might write more on this subject.

Hardware Requirements

To effectively use this tutorial, I suggest you follow along with a PC to test out some of the code I present. A minimal PC with a 233 MHz processor and 32 MB of RAM (or any computer produced after 1998 or so) should be able do everything in this tutorial, but note that some actions, such as compiling, can take much longer (several minutes). You will need some free hard drive space; about a 50 to 75 MB should do it for the C++ compiler, project files, and any other files.

Overheard: Is it possible to put Windows 95 on a Commodore 64?

Some things just don't work to well if you don't have a capable computer!

Operating System Requirements

I will use the widely popular Windows Operating System (OS), mainly because most people know how to use it. The version doesn't matter much, as long as you have Windows 95 or above. However, I know many Linux fans, and I have therefore modified this tutorial to work with Linux as well. So either way, you will learn how to build a compiler on your platform. I do, however, assume that Linux users know their way around and can compile and develop on their own.

Unfortunately, Mac users won't benefit from this tutorial nearly as much because I don't have a Mac and cannot develop a compiler on a Mac. You can try emulation software such as Virtual PC to follow along.

Software Requirements

No, you don't need to guard your wallet. If you have a commercial C++ compiler, such as Microsoft Visual C++, I heartily recommend using it. However, for those of us who prefer to keep our hard-earned money, follow my lead: use Dev-C++, a free C++ compiler with a great IDE (Integrated Development Environment). It might not have the caliber of commercial software, but it sure does the job quite well, especially considering the price. In the next chapter, I will show you how to download and set it up.

Notations Used

I will use several notations throughout this tutorial. The first is to present all code in a `monospace font`. This includes code listings and snippets of code. In addition to a `monospace font`, I will use syntax highlighting with the code listings to make it easier to read and understand. For those not familiar with syntax-highlighting, here is a sample:

Without Syntax-Highlighting:

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


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

```
{
  //Add 2 + 2 and the wait for user
  cout << "2 + 2" << 2 + 2 << endl;
  system("PAUSE");
  return 0;
}
```

With Syntax-Highlighting:

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

int main(int argc, char *argv[])
{
  //Add 2 + 2 and the wait for user
  cout << "2 + 2" << 2 + 2 << endl;
  system("PAUSE");
  return 0;
}
```

Symbols Used

I remember the first programming book I ever read: "Absolute Beginner's Guide to C" by Gerg Perry. It had these boxes with icons of skulls and bombs to indicate when trouble was not far off. I, in my ignorant mind, thought that my computer might actually break down or something. Happily, I learned quickly to the contrary, but I still like the concept of icons that designate special topics. So I'm including some of my own:

Forked Paths

Forked paths boxes will offer insights on why I chose to use one method over another. This is the rare occasion where I will discuss in-depth the pros and cons of doing things one way versus another way.

Explore Further

Explore Further boxes encourage you to do some experimenting on your own. I might give you a puzzle to solve, show you an interesting feature to implement, or ask a question and leave it up to you to find out. Once you have the topic I'll give you a couple of tips, and leave you to it.

Helpful Hints

Helpful Hint boxes offer advice or hints about anything and everything. I might offer a tip on how to accomplish something more easily. I may recollect some things I found helpful when I first started. Or you may just find random ramblings that may or may not pertain to your situation.

Watch It!

Watch It! boxes issue dire warnings about the consequences of carelessness in an area. Here I will embarrass myself by recording some of the things I did wrong. I will also discuss some possible bogs that you don't want to get caught in. Pay close attention to these boxes to avoid nasty pitfalls.

Other Resources

Other Resources boxes will give you suggestions on other reading or research material you might find helpful. To help balance out subjects I don't cover that well, I will at least point you in directions where you can learn what I omitted. Books, Internet websites, and online tutorials might worm their way into these boxes.

My Website:

My website at www.BrianCBecker.com contains much more information on programming, compilers, and so forth. A quick list of items of interest that you can find:

- This tutorial in different formats for printing, online viewing, or downloading.
- The packages of source code and examples that accompany each chapter.
- Other tutorials and resources to continue your compiler education.
- A contact form where you can e-mail me questions, comments, suggestions, etc.
- Forums where you can discuss compilers and compiler technology

Conclusion

That's it for the introduction. You now know where I am heading and what you will learn along the way.

In Chapter 2, you will get a better feel for compilers, their components, and selecting the right tools. As much as I would like to jump right in and start building compilers, it isn't always best to do so without any preparation. By the time you finish Chapter 2, you will be all ready to start working on creating compilers.

In fact, at the end of the chapter you will be ready for anything those compilers might want to throw at you: tricks, monkey wrenches, or even grenades. OK, nothing short of armor and military training can prepare you for grenades, but you should be in good shape to actually start fooling around with compilers.

I also realize that I might have some relatively new programmers in the audience. So I will also take some time to explain some of the components of a compiler system and go into detail setting up a C++ compiler, such as Dev-C++. For those familiar with C++ programming and how compilers work, skimming the chapter should be enough to get you started.

I hope to see you next time with Chapter 2, where we get ready to start doing some work!

Chapter 2

Gearing Up

Welcome back to the second chapter of my Compiler Tutorial!

In the Introduction, I gave you an overview of this tutorial and some basic requirements.

Here in Chapter 2, I will show you how to prepare to start building a compiler. I will cover exactly what a compiler is and how it differs from an interpreter. Additionally, I will guide you through the typical compile process. Finally, I will show you how to get setup to start developing your own compiler.

By the time you finish this chapter, you should be able to:

- Explain the differences between a compiler and interpreter.
- Understand how a compiler, assembler, resource compiler, and linker all work together to produce programs.
- Research and select the right tools for the job.
- Choose the right language to write your compiler.
- Obtain, install, and configure all your tools.
- Be ready to start writing a compiler!

This chapter prepares you for the next chapter where you will learn to build a complete compiler. If you are already familiar with this material, feel free to skim this chapter and then proceed to the next.

Introducing KoolB

Now that you have decided to continue reading my tutorial, I'll introduce you to your mascot for the book: KoolB, your very own example compiler.

In order to actively show you how to build your own compiler, I'm going to show you how I personally built my own KoolB BASIC compiler. This way I can teach you not only how to build a compiler, but give you a practical example that you can use and extend on your own. Instead of grunting and pointing in every direction, I can show you exactly what I mean. You can even copy my code for learning or as a basis for your experiments in compilers.

So say hello to our mascot, KoolB! You will become very familiar with KoolB over the course of this tutorial.

Helpful Hints

Naming Your Compiler

Giving your compiler a name is a very important part of compiler development. Go for a name that is:

- Short
- Memorable
- Distinctive

For example, I'd have a tough time trying to remember a name like MyVeryFirstCompiler, even if it is distinctive.

Likewise, naming your compiler QuickBASIC or QBasic probably isn't a good idea either. Somebody searching for your compiler on a search engine will find thousands of references to totally different compiler! So think carefully about what you want to present.

For the record, KoolB was a name I picked out of the air. Originally, the name was Kool-Bee, but KoolB looks so much kooler. Also, a search on Google turned up only a handful of links. So the name KoolB does meets all three requirements.

Simply KoolB

To give you a taste of building compilers, I will show you how to build a very simple, yet complete compiler in the next chapter. This will not be the real version of KoolB, because that will take more than a single chapter. This compiler will accept only several commands, but will compile real programs. Because it is so plain, a good name is Simply KoolB.

I would love to jump right in and start building Simply KoolB. Unfortunately, as in many projects, you first need to go through a planning stage. This is exactly what this chapter is. I will show you all the tools you need to build a compiler like Simply KoolB and then how to set them up.

For readers who are just getting started in building compilers, I suggest you follow along as if you were building Simply KoolB yourself. That way you get hands on experience. Then once you feel more comfortable with developing compilers, you can start adding some of your own features to Simply KoolB (and later, the KoolB compiler).

For readers who are more advanced and have already dabbled some in compilers, I suggest you follow along and use this chapter to get ready to start building your own compiler. Also, feel free to use KoolB as a base for your compiler.

Compilers vs. Interpreters

The first order of business is to decide whether Simply KoolB or any programming language you design will be a compiler or an interpreter. Both have pros and cons, and often the decision is a choice of personal style (unless of course your boss or professor has a different idea).

When programming anything, you start out with a text file containing the programming language (like BASIC or C++). This text file is referred to as the source code. But the computer can't understand this source code. For example, when you write a program, you can't just throw it at the computer and say: "Here, now run it!" Why not?

Because the computer doesn't understand English. To test this, try telling your computer to turn on when you get up in the morning. No, nothing happens (and if it does, I'd be a little frightened!). Computers don't speak our languages. So you need a compiler to translate your source code to the language your computer understands.

The language the computers understand is binary language, or long series of zeros and ones. It is definitely not a pretty sight for humans to read, but that's what the computer likes. This binary language is stored in a file called a program and is just a bunch of instructions telling the computer what to do (just like your mother used to do).

An interpreter does the same thing in a bit different manner. Instead of translating the source code to a program that the computer can understand before the program is run, the interpreter translates the source code as the program runs. For instance, whereas a compiler might convert 2 + 2 to

binary language and then tell the computer to run it, an interpreter looks at 2 + 2 and then manually adds the numbers together.

The classic example of this is a foreign language translator, such as Spanish to English translator. In the example of a compiler, the foreign language translator will spend an hour translating an entire document from Spanish to English. Then any English speaking person can read the paper.

In the example of an interpreter, the foreign language translator will wait until somebody needs to read the document. Unfortunately, the reader cannot read Spanish, so the foreign language translator has to sit down and read him the paper, translating it to English as he reads.

As you can see, the compiler translates everything all at once whereas the interpreter translates as parts as needed. The main disadvantage to an interpreter is that the interpreter must always be available to translate to anybody at any time. The main disadvantage to a compiler is that sometimes a lot of time is needed for the initial translation.

So which approach should you take? In truth, it depends on your needs.

Compiler Advantages:

- You get faster, more compact programs
- You learn the internals of computers
- You have a more professional programming language

Compiler Disadvantages:

- You have to learn & use assembly language
- You don't have as much control over debugging
- You have to put in much more work to make it cross-platform

Interpreter Advantages:

- You can write it faster because of less complexities
- You can debug it faster
- You can write cross-platform interpreters easily

Interpreter Disadvantages:

- You get slower programs
- The interpreter must always be distributed with the application

If you look at this list, you might be tempted to expect that Simply KoolB will be an interpreter and that I will concentrate on building interpreters. That is not the case, however.

Why not? After all, it seems easier overall. The main reason is that compilers have a greater status than interpreters because they generate much faster and more efficient programs. Also, you will learn more.

Other Resources

Learning About Interpreters

However, if you really want to try your hand at an interpreter, I suggest you check out Steve's ***Let's Build A Scripting Engine***. He will guide you through the process of building a BASIC interpreter from the ground up. You can find his tutorial in the Files section of the QDepartment (http://groups.yahoo.com/group/QDepartment/Files)

Explore Further

Other Types of Compilers

Actually, many more categories of compilers exist. You may want to learn more about these on your own time. I'll get you started with a brief description of several:

- **Translator:** Translates from one language to another. For instance, from BASIC to C.
- **Byte-code Interpreter:** Translates the source code from English text to a compact form of code known as byte-code or P-code. This allows the interpreter to run faster. Examples include the popular Java
- **JIT compiler:** Stands for Just-In-Time compiler. First, a compiler translates a program to a byte-code. But instead of interpreting this byte-code, when the program is run, a compiler translates the byte-code to native computer language just in time (hence it's name). Microsoft's .NET platform is based on JIT compilers.

The Compile Process

In order to create a compiler like Simply KoolB, you first must need to know how the compile process works. Different compilers will use differently

process (often radically different). This approach is not the best nor the easiest, but many compilers use it. So I will briefly go over it here.

Obviously, computers can't understand English, or even our programming languages. Actually, I take that back…to some people it isn't so obvious:

User: "Hey, can you help me? My program doesn't work."

Consultant: "What is the problem? Are you using Turbo Pascal?"

User: "Yes, the program just blocks the machine."

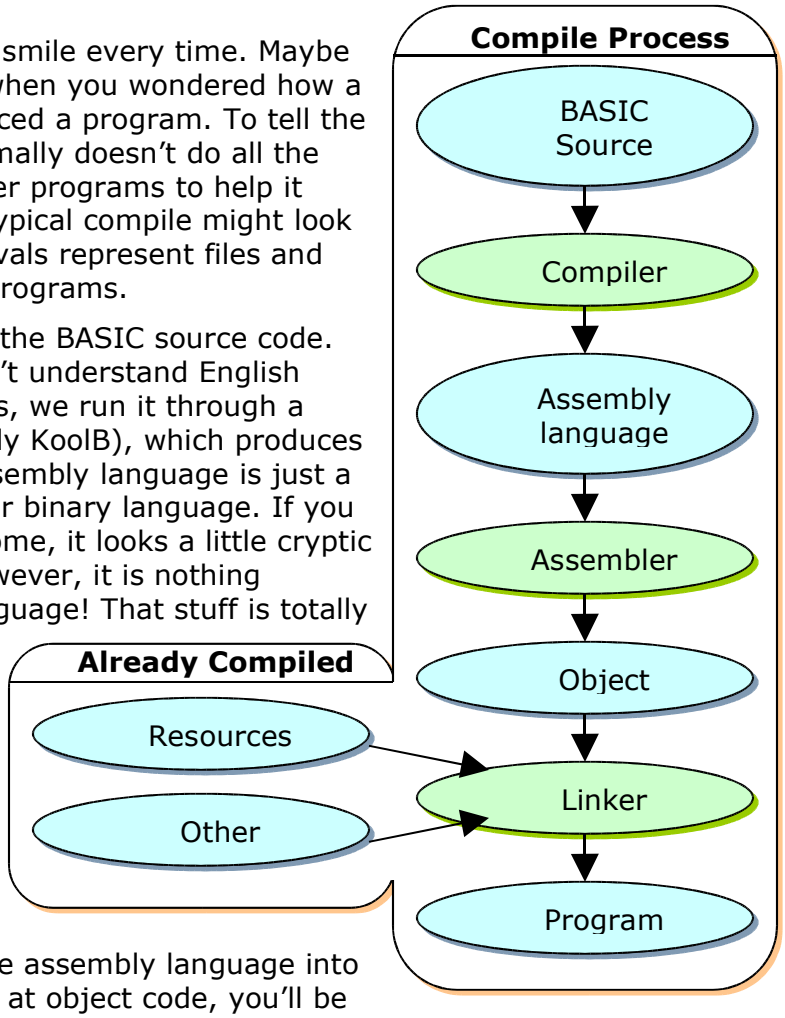
Consultant: "Well, does it compile?"

User: "I don't know -- it just doesn't run. You see? There's the EXE file. If you run it, it blocks the machine."

Consultant: "And where is your source, the PAS file??"

User: "I wrote it and renamed it to EXE so it could run."

Figure 2.01



The last line makes me smile every time. Maybe you remember a time when you wondered how a compiler actually produced a program. To tell the truth, the compiler normally doesn't do all the work. It uses some other programs to help it along. For instance, a typical compile might look like Figure 2.01. Blue ovals represent files and green ovals represent programs.

First, you start off with the BASIC source code. Since the computer can't understand English programming languages, we run it through a compiler (such as Simply KoolB), which produces assembly language. Assembly language is just a text version of computer binary language. If you have every looked at some, it looks a little cryptic (OK, a lot cryptic!). However, it is nothing compared to binary language! That stuff is totally unreadable. Unless you have a a strange, special talent I don't know about.

Of course, the computer can't run assembly language either. So another type of compiler called an assembler translates the assembly language into object code. If you look at object code, you'll be scratching your head because object code is not readable by humans. It is actually very close to native computer language, but not quite.

Actually, this object code represents only the source code you compiled. But you might want to add other resources (such as pictures, icons, other files) or

libraries to your final program. So the final step is a linker, which takes all these files and combines and links everything together to form your final program (hey, maybe that's where the name came from).

The linker also makes sure the program will run on your operating system. For instance, in Windows, it will link your program to the Windows DLL (Dynamic Link Libraries), such as

- Kernel32 for core Windows routines like managing memory
- User32 for windows, buttons, and a user interface
- Msvcrt for C library routines
- GDI32 for drawing

On Linux, you will be linking to the standard C libraries.

So now you know how a basic compile process works for Simply KoolB. It might seem a little confusing at first, but just think of it as a team effort. Everybody (the compiler, assembler, and linker) works together to make sure that your BASIC source code gets turned into a program.

Forked Paths

Skipping the Assembler and/or Linker

Actually, you can actually skip the assembler by writing object code directly. However, you will find out that this will be harder than assembly language. Now if you like to punish yourself even more, you can go directly from the BASIC source code to a program. This would be even harder. The advantage to this is that you can speed up compile times by 5 to 100 times. This might be something to experiment with it, as you get more advanced.

What Languages?

In order to build a compiler, you need to choose two programming languages:

1. To build the compiler in
2. For the compiler to compile

In both cases, you pick the language. That's one great thing about building compilers: you get to dabble with any programming language you want. So go wild!

In the first case, you typically want to pick a language that has been around for a while and is pretty complete. Many choose some type of C, whether it be what one friend calls "vanilla" C, C++, or Microsoft's new C#. Other

popular languages include Java (you could write a compiler applet even), although it is slower than most languages.

Internet languages like Perl or PHP have been used. I've heard of specialty languages like Python being used as well. In fact, I've even had some success with more basic languages like, well, BASIC! Just make sure the language has good string handling capabilities.

The second language will be the programming language that your compiler will use. You pretty much a have a free reign in this case. You can clone a good language that has been abandoned. You can improve upon a good language. Or you can develop your own programming language. It is totally up to you!

To appeal to the widest audience, I will use C++ to write KoolB (and Simply KoolB). I chose C++ mainly because it has long been a standard for professional programming. I also wanted to learn C++ at the time. However, I won't use all the advanced features of C++, so if you've been a long-time C programmer, don't worry, just brush up on some of the object and classes terminology. For others that may have been using another language such as Visual Basic, I mentioned some good tutorials in the last chapter. These will get you programming in C in no time!

KoolB itself will be a BASIC compiler. Almost everybody remembers dabbling in either QBasic back in the day or have used Visual BASIC, so you should be familiar with the BASIC syntax. If not, you can find many free BASIC compilers or interpreters on the Internet that you can try out.

Explore Further

Self Compiling Compilers

A funny thing happens when you choose the same language for both the first and second case. You wind up with a compiler that can actually compile itself! This is because the language the compiler is written in is the same language the compiler compiles!

So eventually, you will get to a point where you can develop the compiler with itself. This is called a self-compliable compiler.

For instance, if you want to make a BASIC compiler, you can write the compiler in a BASIC dialect that is as close to the BASIC language you want to make. If the language isn't totally compatible, you can still do it if you are willing to do some re-writing of the compiler.

This is an interesting concept that is often referred to as bootstrapping, because you are basically pulling yourself up by pulling on your bootlaces.

Making a Tool's Inventory

Before you can get started, you need to install and set up some tools. But what tools do you need to start to develop compilers and specifically, Simply KoolB? Obviously, a computer running Windows or Linux would be a start in the right direction. But how about some others:

- **C++ Compiler** to write the compiler
- **Assembler** to assemble the output of the compiler
- **Resource Compiler** to compile resources into object files
- **Linker** to link the object files to programs

And, of course, every compiler developer needs the Developer Kit: Aspirin, Maalox, and Cyanide. I could sure use Aspirin item after hours of debugging! I never did need the last two though. Keep them handy…you never know.

I'll examine each component in detail. In most cases, several options will exist for each tool. For example, you will find many free & good assemblers.

I'll guide you some steps I went through to choose these tools for KoolB and Simply KoolB. You can use these same steps to find other tools for your own compiler.

A C++ Compiler

First, you need a compiler to build your compiler with (what a novel idea). For Simply KoolB, that would be a C++ compiler. For your compiler, that might be completely different. However, for now, I urge you not only to read along, but to also go ahead and actually follow along by installing the software and testing out my KoolB code (who knows, you might find a bug).

Choosing a C++ compiler probably is the easiest decision you need to make. If you have a commercial suite that you like (Microsoft's Visual C++, Metrowork's CodeWarrier, etc), by all means stick with it.

Otherwise, if you don't have a C++ compiler and would rather not hear that sucking sound on your wallet (otherwise known as cash drainage), follow my lead: use Bloodshed Dev-C++. It is completely free. You can just download it off the Internet. Of course, being free, you can't expect it to do all the things that a commercial package would offer. Surprisingly enough, it does quite well. I have developed two entire compilers with it and not had any serious problems.

To cater to the lowest common denominator, I will be describing and using Dev-C++ in this tutorial. If you have a commercial C++ compiler, I assume you already know how to use it. If not, I'd suggest installing Dev-C++ and following along.

If you need help with installing Dev-C++ from http://www.bloodshed.net, please see Appendix A for a step by step instruction guide.

For all those Linux users out there, I assume that if you are using Linux, you are already something of a power-user and probably use some esoteric software like Emacs or vi. So the only subject I will cover will compiling with GCC via the command line. Other than that, you are on your own!

Forked Paths

Other Free IDEs

(**I**ntegrated **D**evelopment **E**nvironments)

Actually, there are quite a few other free IDEs out there. You can use a general purpose IDE or a specialized one. Doing some research on the internet will yield a good number of C++ editors. A notable one is MinGW Developer Studio at http://www.parinya.ca/. It may rival Dev-C++ in the future (if it remains free).

Of course, you can always use Notepad and the command line GCC to develop your compiler, but most of us like something a bit easier.

Creating Space for KoolB

Before I guide you through the process of installing and configuring all the rest of the tools you need, you first need some space to develop your compiler. I suggest you create a "Compilers" folder under your C drive (or your /home directory for those using Linux). If you decided to organize your folders differently, just adjust the instructions in this tutorial to match your folder locations.

Once you have a root Compilers folder, create some sub-folders to put everything into. For example, my folders are setup like so:

C: Drive (or /home directory)
- Compilers
  - Tools
  - Resources
  - KoolB
  - Simply KoolB
  - Documentation

Modify things as you see necessary. For instance, a Temp folder might be nice to have. Organizing things this way helps you to be able to find things when you need them (which is definitely a plus).

Choosing an Assembler

If you do a quick search on the Internet, you will find quite a few assemblers. So how do you choose which one to use?

First, I would make a list of requirements and things that you need the assembler to be able to do. For instance, do you need an assembler that runs on different operating systems? Or is it important that the assembler run on a computer with little memory? Sometimes it is very easy to choose because there is only one choice! In another instance, you might have used a particular assembler in the past, so it doesn't make sense to relearn another one.

But what do you do when more than one assembler meets you requirements? I'll take Simply KoolB as an example. My list of requirements looked like this:

- Must run on both Windows and Linux.
- Must be able to make dynamic link libraries, or shared libraries on Linux.
- Small size was preferable

That was about it for requirements. So very quickly, I dropped several DOS assemblers from my list. Assemblers that only ran one OS got crossed off too. That left basically two assemblers on the list:

- Flat assembler @ http://flatassembler.net/
- Netwide assembler @ http://nasm.sourceforge.net/

So what do you do? Download them and try them out, of course. As you learn more about each one and as use them, you should be able to find one that suits your project the best.

In my case, the best choice (in my opinion) was the Netwide assembler. It was more established and was available on more operating systems than Flat Assembler. However, this has and is changing, so by the time you read this, you need to evaluate everything for yourself.

And if you just can't decide, flip a coin!

Once you have settled on an assembler, you need to install it. In the case of the Netwide Assembler, just download the latest Windows binaries (at the time of writing the latest is 0.98.38). Put it in your Simply KoolB folder and extract the zip file. If you don’t have any unzipping software, you can download 7-zip for free at http://www.7-zip.com. The actual assembler is named nasmw.exe. The other program, ndisasmw.exe, is a dis-assembler and can be safely deleted.

While you are at it, download the documentation and store it in your Resources folder (I’d create a new folder called NASM Docs or something). This will greatly ease much head pain because you won’t have to bang your head against the wall as much. You can just read the documentation and (hopefully) discover what went wrong.

Choosing a Linker

Choosing a linker isn't as difficult as choosing an assembler because you are more limited. Basically, the linker just combines multiple object codes into a final program. Since program formats are different on different operating

system, you typically need to find a linker for each operating system your compiler supports.

The linker also links system functions to the right libraries. The operating system provides many functions so programmers don't have to constantly rewrite commonly used functions. On Windows, these functions are called WinAPI calls (short for Windows Application Programming Interface). In Linux, you can use the stdc libraries (short of the Standard C Library).

For Linux, the standard linker comes as part of the GCC development package (THE C & C++ compiler for Linux). So in this case, your choice is a no-brainer.

Windows is a bit more complicated. In looking around, you will see that a lot of linkers use large LIB (library) files. The linker uses these files to match system functions with the right DLL. However, this can take up many megabytes of information (although you can minimize this through the use of selective libraries and compression).

So when I found a brand new linker named GoLink that didn't use these LIB files, I got excited. GoLink was written by an assembly fanatic named Jeremy Gordon. By actually loading and reading the system DLL files, this linker could determine which functions to link to what DLLs. That certainly sounded like a good deal to me, so I definitely used it in KoolB. However, you might fancy another linker.

To install GoLink, visit http://www.godevtool.com/ and download the latest version (the latest at the time of writing is .22) to your Simply KoolB folder and extract the contents of the zip file. Documentation is included with the linker, so go ahead and move all the files except the GoLink.exe program to your Resources folder (as you did earlier, create a new folder named GoLink Docs for these files to go in).

Choosing a Resource Compiler

For Windows, an additional tool is needed: a resource linker. Windows allows you to include different types of resources in your final application. What types of resources can you include? Typically, you can include icons, images, menus, dialogs, and even raw files. To get these resources included in your program, you have to convert them to special object code using a resource compiler.

Again, choosing one isn't difficult as there aren't too many available. Since the author of the linker KoolB uses also developed a resource compiler named GoRC, it seemed simplest to use that. However you make your own decision.

The install procedure is very much like GoLink. You can download GoRC from http://www.godevtool.com (the latest version at the time of writing was .81) to your Simply KoolB folder. Unzip the contents and one more time create a new folder named GoRC Docs in your Resources folder and move all the files except GoLink.exe into that folder.

Conclusion

Are you ready to start building Simply KoolB? Me too! I've showed you a bit about gearing up and getting ready to start building compilers. I've also covered some of the tools a typical compiler needs to help it along. I now pronounce you ready!

So what are we going to do in the next chapter? For starters, I will show you how to build a very plain compiler named Simply KoolB from the ground up. Completely! It will function just like a real compiler, except of course it will only accept a handful of commands. But that's just the appetizer. You will also learn a little bit about GUI programming I show you how to create a simple IDE/editor for Simply KoolB. As an added bonus, I'll show you how to do some cool stuff like package and set up an installer for our premier software!

So put on your learning caps and get ready to work next chapter. See you then!

Chapter 3

Simply KoolB

Welcome! So are you ready to learn how to build a compiler from scratch? Good, because that's exactly what I'm about to show you how to do. In the last chapter, we reviewed some compiler building basics and got ready to get down and get some coding done. That's precisely what we are going to do here in this chapter.

In fact, this chapter is almost a mini Compiler Tutorial in itself. Before we dig into some of the technicalities of building compilers, I thought it would be fun to fool around building a very small, but complete compiler. It won't be as complicated as KoolB (that'll take the entire book to complete!), so an apt name is Simply KoolB. I'll give you a very brief tour of building compilers, showing you how to to build Simply KoolB in the process. You'll get familiar with most of the aspects of a compiler, and have fun doing it!

As an added bonus, I'll be covering not only creating just the compiler, but many other important aspects necessary to a successful compiler project: an IDE for usability, a website for PR, & an installer for professionalism.

This is definitely a follow-along chapter, so dig out your C++ compiler (see Appendix A for help downloading and installing a free one from the Internet).

Ready? Let's get started!

First Stop: Planning

Without a doubt, planning is boring. But it is necessary. Yet that still doesn't make it any less boring. Since this is a small project, let's keep the bureaucracy to a bare minimum, OK? The shorter and sweeter, the better.

So what are the essentials a user would want out of a compiler? How about a list:

- A compiler
- An editor of some sort (a simple Integrated Development Environment or IDE for short)
- Some documentation
- Some PR via a small website
- A nice installer

Anything less than that and you'd have some disgruntled users. So that's the game plan for the rest of this chapter. Of course, the compiler and the editor is the most important, so let's focus on those first.

Creating the Programming Language

Before we go running around nilly-willy coding up anything and everything, first let's take a second to put down on paper what we want our language to look like. No doubt it will change as we code, but this gives us a place to start. As I mentioned earlier, Simply KoolB will be a BASIC sort of language, as that is a very familiar language (hey, if my 70+ grandfather remembers learning BASIC, you can tell it's been around).

In BASIC, each line is it's own statement, so all we have to do is figure out which statements we want. So what are some easy BASIC statements to build into Simply KoolB? I've thought of seven:

1. Blank lines
2. REM to comment out a line of code
3. CLS to clear the screen
4. PRINT “STRING” to print out a sentence onto the screen
5. RUN “COMMAND” to run an external command via the OS command interpreter
6. SLEEP # to sleep a certain number of seconds
7. WAIT to pause the program until the user presses ENTER

OK, so I lied. Technically, you could argue that there are only six commands here as blank lines don't really count. But that's the plan: to create a compiler so you can program with these seven commands. A sample program in Simply KoolB might look something like this:

```
REM         A test program for Simply KoolB by Brian C. Becker
REM         Part of the Compiler Tutorial (www.BrianCBecker.com)
REM
REM         Simply KoolB is your example compiler
REM         written in less than 1000 lines of code!

REM         Print out a welcome!
PRINT       "Welcome to a demo of Simply KoolB by Brian C. Becker!"

REM         Wait a couple seconds before clearing the screen
SLEEP 4
CLS

REM         Ask the system to list the contents of the current
directory
PRINT       "Printing the folders in your C folder:"
RUN         "DIR C:\ /a:d /w"

REM         Wait until the user hits ENTER
PRINT       "Press ENTER to view a goodbye:"
WAIT

REM         Clear the screen
CLS

REM         Print goodbye
PRINT       "Goodbye!"
```

As you can see, you can't program really cool programs in Simply KoolB, but it isn't a total waste either – you can create some small programs with it. You might say that Simply KoolB is a very small "toy" compiler. But when you see Simply KoolB turn this simple BASIC into a 2 KB Windows executable, you'll feel proud of of what you accomplished.

Getting Started: Creating the Project

Now that we have an idea of what we're about to do, let's go do it. We'll start with coding the compiler as that is the most important. So go ahead and launch Dev-C++ and create a new project by going **File -> New -> Project**. Select a **Console** project, make sure the language is **C++**, and then name the project "KoolB". When prompted where to save the project, save it to "C:\Compilers\Simply KoolB\Source\Compiler" (or where ever you want). By saving your source in a separate folder, you can prevent all those *.cpp files and other programming related files from cluttering up the main directory.

The problem with this is that when you compiler KoolB, the executable gets placed in that same folder. But that is rather inconvenient, so this is what we will do. Go to **Project -> Project Options** and click on the **Build Options** tab. In the **Executable output directory**, type "../../" This tells Dev-C++ to put KoolB.exe two folders up, or in the "C:\Compilers\Simply KoolB" folder. That's exactly what we want, so click **OK**.

Dev-C++ already created a very simple C++ program for us. However, we don't need the first line (`#include <iostream>`), so delete it. After you do this, you should have this C++ code:

```
#include <stdlib.h>

using namespace std;

int main(int argc, char *argv[])
{

	system("PAUSE");
	return 0;
}
```

You can compile this by clicking Execute -> Compile. When asked what to save the file as, just choose the default filename of "main.cpp" and click OK. If all went well, Dev-C++ should have delivered a KoolB.exe to "C:\Compilers\Simply KoolB." If you double-click it, you will notice a black console box will pop up. Because compilers don't really need a fancy interface, a console (DOS-like interface) is just fine. In fact, even commercial compilers like Visual C++ or VB.NET have console compilers. However, they have very fancy editors and Integrated Development Environments (IDEs). Not to be outdone, we'll also create an IDE for Simply KoolB later in this chapter.

Now that we've compiled it, I'll quickly go over it for those a bit new to C++. Our `#include` statement includes the part Standard C Library so we can use it. The `using namespace std;` line tells the compiler to use the standard namespace, which includes the Standard C++ Libraries (not to be confused with the Standard **C** Libraries).

The **`main`** function is where everything all begins – when the program runs, it automatically runs the **`main`** function. I'll talk about argc and argv later, but for right now, you can just think of these as pieces of information that the operating system (OS) gives us.

And finally, the `system("pause");` statement pauses the program until you press a key. This prevents the program from flashing a black console box if you double click the program from Windows Explorer. And finally, the `return 0;` statement lets the OS know that the program is done. Returning 0 tells the OS that everything succeeded.

Not to bore readers familiar with C or C++, I will keep my explanations of the C/C++ code to a minimium. If you are having a bit of trouble understanding what is going on here or throughout the rest of this chapter, I encourage you to check out the links in Chapter 1 that will help you learn C and C++.

Requesting the Filename

Before any compiler can start compiling, it first has to know what to compile. This means that somehow the user will need to give us the name of the file to compile. We have two ways of getting this information. One is to use the command line and the other is to directly request it. To cover all our bases,

we will use both. If the user doesn't pass the filename through the command line, we'll ask the user for it. So what is the command line? That's the argc and the argv I was mentioning earlier. When you run a console program, you usually pass it paramters like:

```
DIR C:\ /a:d /w
```

If you typed this in DOS, it would display a list of all all the directories in your C:\ drive. And the /w would list the directories horizontally instead of just vertically (wide format).

Likewise, we can pass parameters to KoolB, such as the the filename of the BASIC source code to compile:

```
C:\Compilers\Simply KoolB>KoolB "Test.bas"
```

If you ran KoolB this way, the operating system would automatically pass two parameters to your main function:

- `argc`:The number of parameters passed to the program. In this case, argc would be 2 (KoolB and "Test.bas")
- `argv`: An array of strings that contains the actual parameters. Of course, this is a C (and C++) style array, so you would access "KoolB" with argv[0] and "**"Test.bas"**" with argv[1]

So the solution is pretty easy. If we have two parameters, we just assume the 2nd parameter is the filename. However, if argc is not two, then we will kindly ask the user for the name of the file.

Forked Paths

Such a Fuss About Strings

When it comes to working with strings, we have two choices. Since compilers do so much with strings, choosing wisely is rather important. We can use C style strings or the C++ STL style strings. C style strings are more low-level and harder to use, but you have more control over them and you can probably eek a bit more efficiency out of them. STL strings are much more similar to strings in other languages, like BASIC; additionally, they are much easier to use. And since we will have to be using lots of strings, I personally think STL strings would be the better choice. But for those who disagree with me and prefer null terminated C strings, just wait until we start mucking with the assembly language. There we have no choice: we have to use C style strings. So we get the best of both worlds, right?

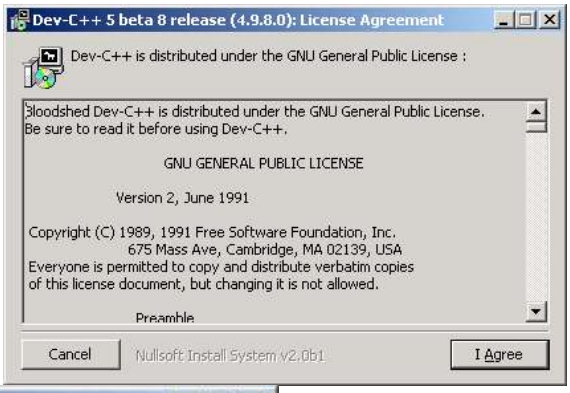
```
#include <stdlib.h>
#include <iostream>
#include <stdlib.h>
#include <string>

using namespace std;

// Global Variables for use in Simply KoolB
string FileName;

// General functions
void RequestFile();

int main(int argc, char *argv[])
{
	// Print welcome
	cout << "Welcome to Simply KoolB by Brian C. Becker" << endl;
	cout << "Written for the Compiler Tutorial (www.BrianCBecker.com)."
	<< endl << endl;

	// Check for an existing filename
	if (argc != 2)
	{
		RequestFile();
	}
	else
	{
		FileName = argv[1];
	}

	system("PAUSE");
	return 0;
}

void RequestFile()
{
	char Temp[1024];

	// Ask the user for the name of the file to compile
	cout << "Please enter the file to compile: ";
	cin.getline(Temp, 1024);
	FileName = Temp;

	cout << endl << endl;
}
```

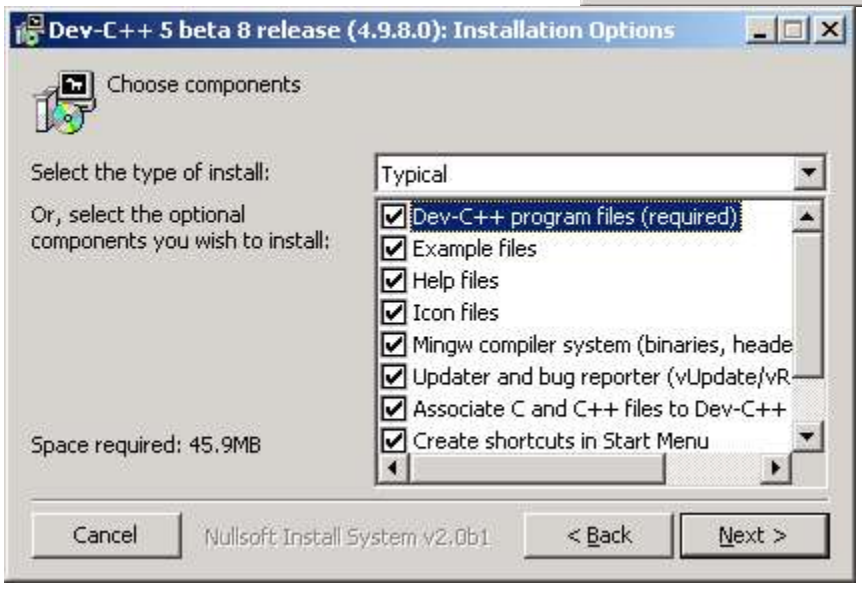
Installing Dev-C++

Right now, I'll show you how to download and install Dev-C++. If you already have a C++ compiler setup (whether it be Dev-C++ or a commercial suite), feel free to skip this section. Otherwise, make sure you are on a Windows computer with an Internet connection.

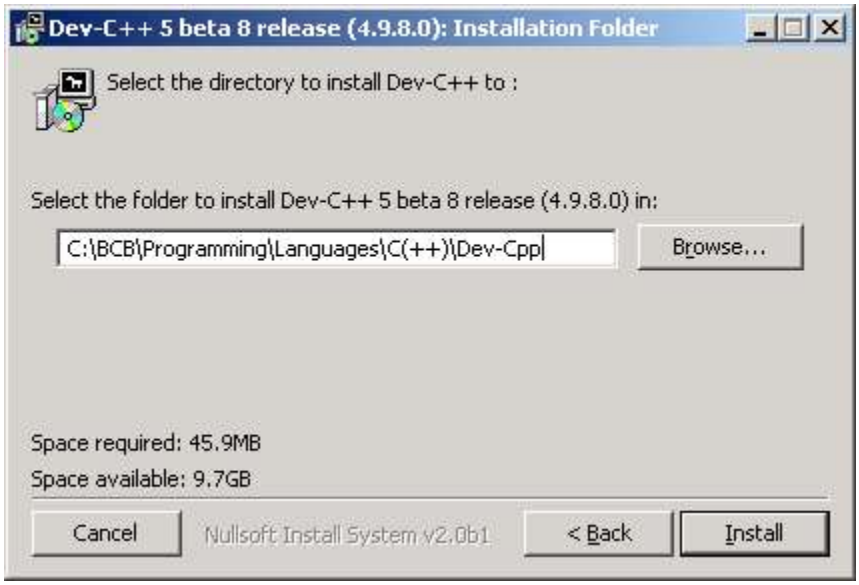
First, download the latest version of Dev-C++ (version 5 beta 8 or 4.9.8.0 at the time of writing) from http://www.bloodshed.net. The full download includes both the C++ IDE (editor) and the C++ compiler. Run the setup file:

Click **OK** to continue to the main installation. Of course, you want to read the license agreement, so spend at least 15 minutes scrutinizing it closely (like that would ever happen!). Then click **I Agree** (to work for Bill Gates for 5 lifetime sentences).





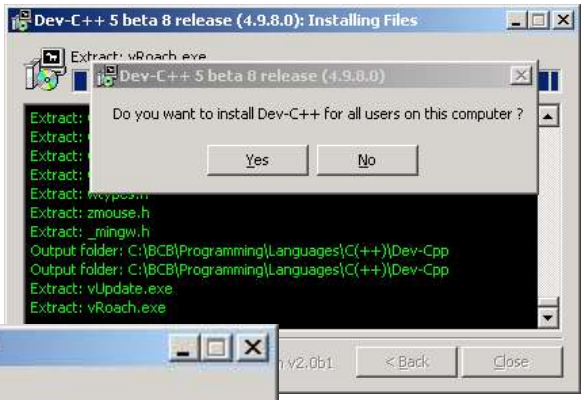
The next page allows you to select what you want to install. Unless you absolutely don't have the space, I suggestion you install a typical set of components (about 50 MB).

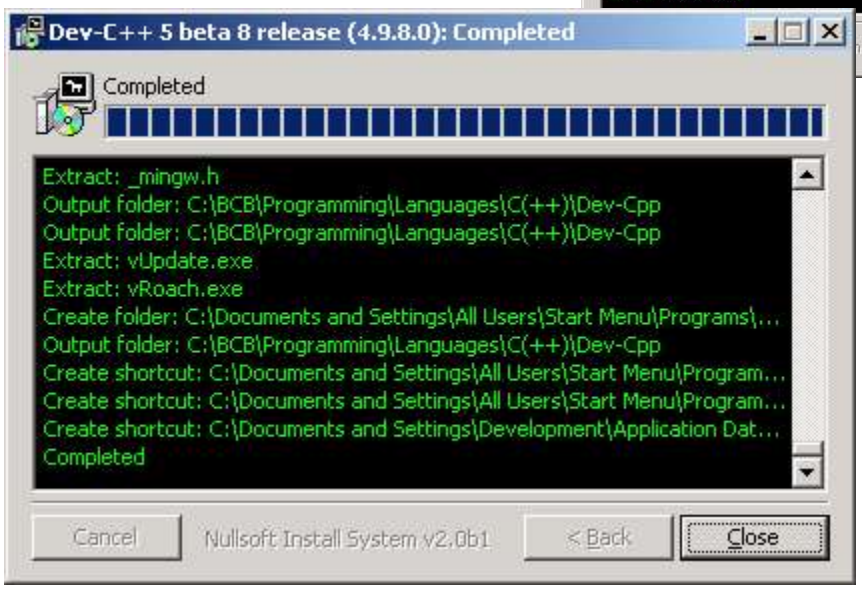


Click **Next >** and select a folder to install everything to (you can just accept the default if you want).

Click **Install** and sit back and relax (or go grab a quick snack) as Dev-C++ installs for you.

After several minutes, you are asked if you want to set up Dev-C++ for all users on the computer. I suggest you answer **Yes**.



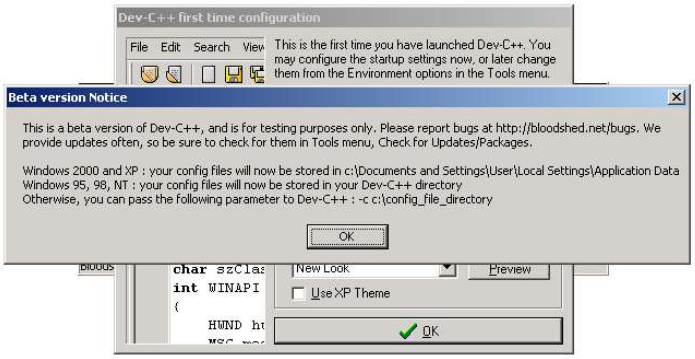


Installation will then complete, so click **Close** (say that 10 times really fast!).

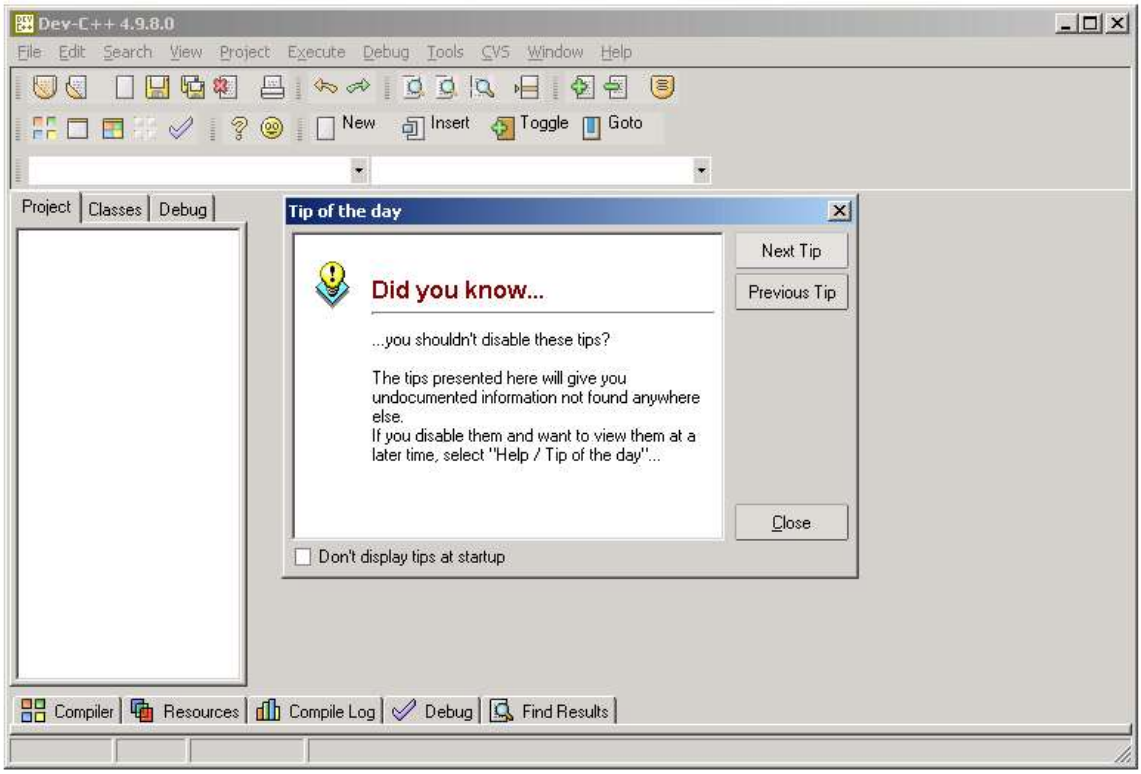
Dev-C++ will launch:

Now you are getting somewhere! Next you will be presented with first time configuration dialog.

Basically, it says that this is a beta release that hasn't been 100% tested and may crash. By the time you read this, development might have stabilized and version 5 might be out. If not, click **OK**.



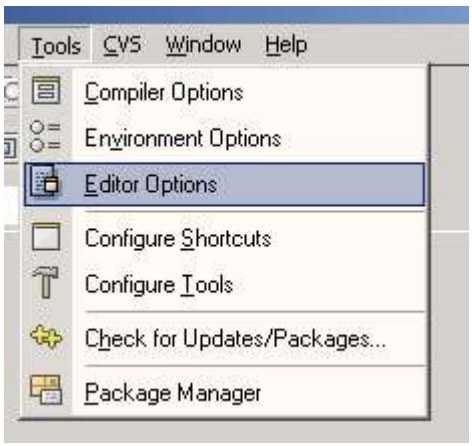
Now choose your language. If you want Dev-C++ to take on the look of Microsoft Office XP, click the '**Use XP Theme'** (personally, I think it is nice). You can also choose a theme if you want. Click **OK**.

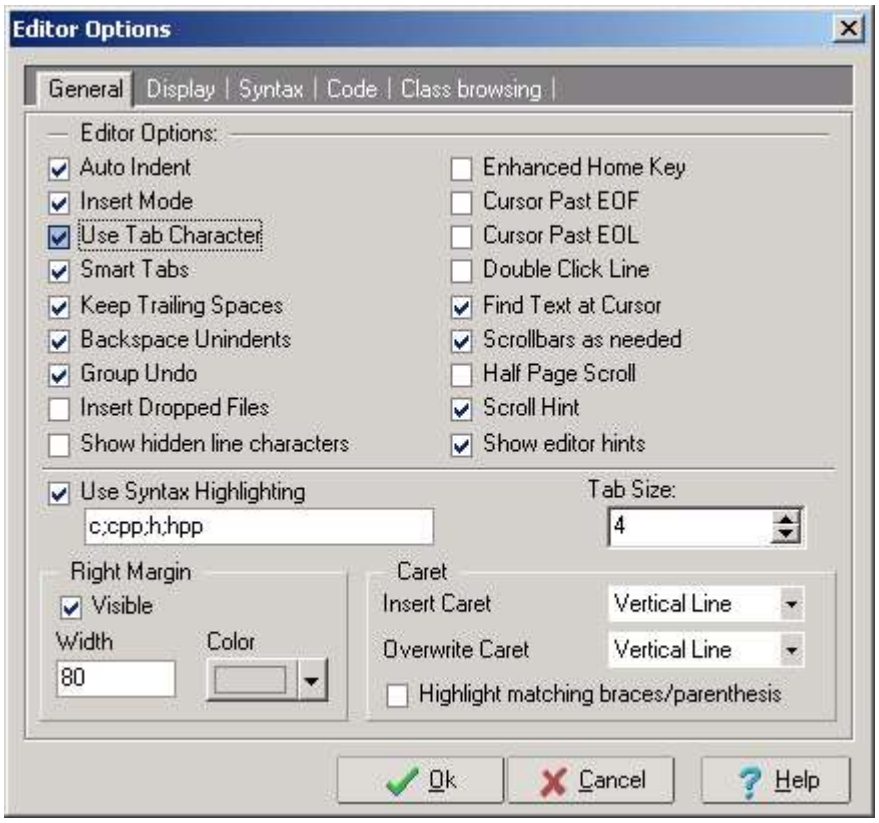


Finally! Dev-C++ is installed:

Click **Close** to close the Tip of the Day (disable it by checking **Don't display tips at startup**).

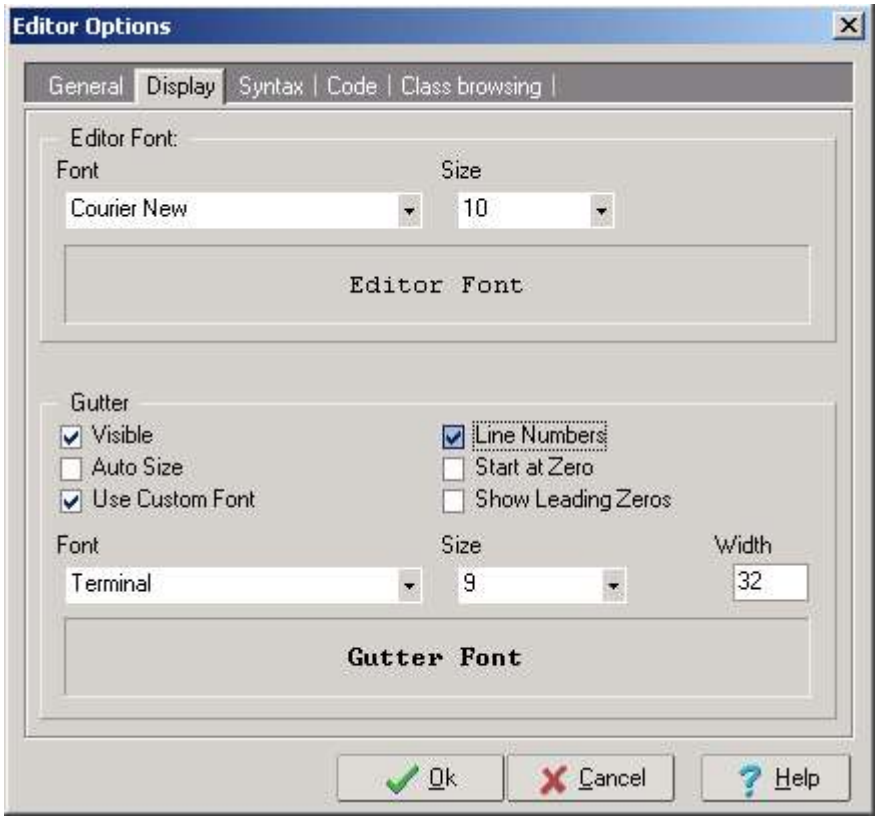
If you want, there are several options that will make your life a lot easier. First, go to **Tools** -> **Editor Options**:





That will bring up a tabbed dialog where you can change various options related to the text editor.

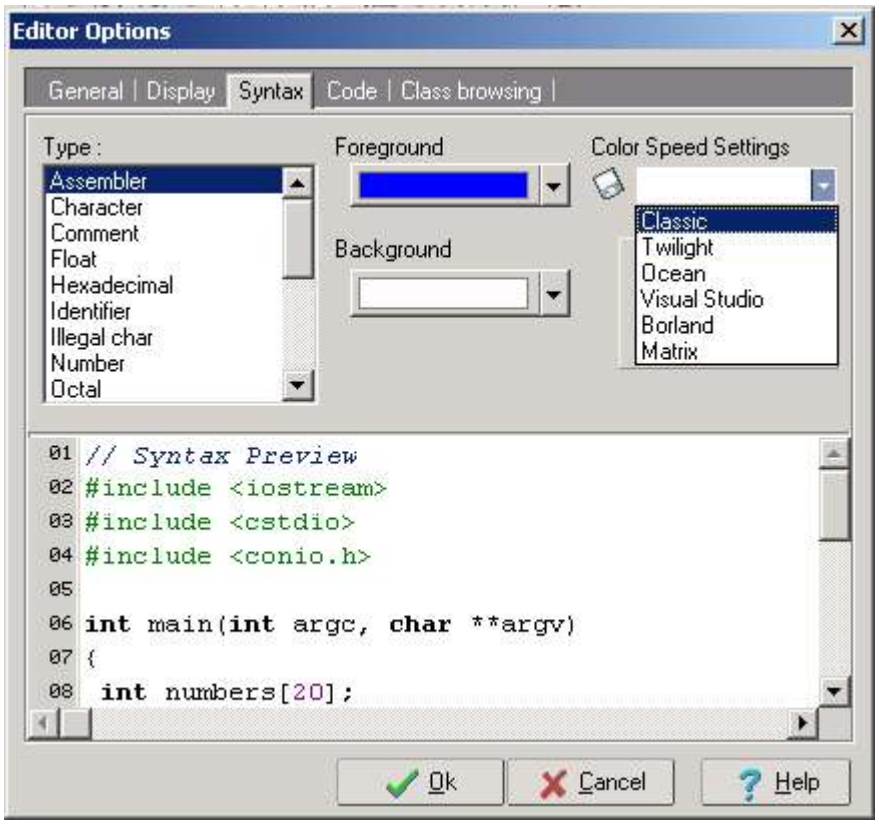
Check the **Use Tab Character** to enable tabs.



Then move to the **Display** tab.

Check the **Line Numbers** options if you would like to see line numbers. I always use them because I just like to know what line I'm on.

Then move onto the **Syntax** tab.



Select a **Color Speed Setting**. I use **Classic**, but some of them are pretty cool as well. Finally, click **OK**.

That's it! Dev-C++ is now ready for use. Of course, there is a lot more for you to explore on your own.