

## Unit 1. C Programming Model

- 1.1 [The Wonder of Program Execution](#)
- 1.2 [The Visual C++ Debugger](#)
- 1.3 [Variables and Addresses](#)
- 1.4 [Data and Function Calls](#)
- 1.5 [The Code Itself](#)

### Assessments

- [Exercise 1: Decoding Lab](#)

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.


## 1.1 The Wonder of Program Execution

Programming is a difficult task requiring a strict exercise of **logic** and a capacity to **organize** substantial amounts of information. Because these challenges require our full attention, we seldom stop to think about all the work that goes on **behind** the scenes when we execute our finished programs. In this module we will try to give you a feeling for how nearly miraculous it is that the programs that we design end up commanding innumerable **devices and mechanisms** in complicated sequences of tiny events. These events, which appear insignificant when considered individually, behave according to the specification of our programs when considered together. When writing a program, you might view yourself as the CEO of a large company who, though unaware of the individual work of thousands of employees, directs their efforts in one coherent strategy. This requires **organization** and a lot of middle **management**, the nature of which we explore in this course.

- [1.1.1 Execution as Physical Process](#)
- [1.1.2 Levels of Abstraction](#)
- [1.1.3 Compilers and Debuggers](#)
- [1.1.4 The Visual C++ Environment](#)

### Assessments


- [Multiple-Choice Quiz \(1\)](#)

Try to remember how difficult it seemed to add two long numbers when, as a child, you were learning addition.  You had to remember tables, a way of organizing the digits, and several rules for "carrying." When each addition seemed a laborious application of an algorithm, who could have guessed that several months or years later you would not even pay attention to the process! You probably went through a similar experience when you learned long division, and many of us have still not completely mastered the algorithm for extracting square roots.

Yet when you write  $c = a + b$  in a programming language, the computer will, at some level, carry out exactly those steps that you thought so laborious. In fact, in this example, the computer will do something else: it will also need to figure out what you mean by  $a$  and  $b$ , since  $a$  and  $b$  are not numbers, but *symbols* whose values are meant to be substituted before the addition starts. In response to this simplest of instructions, then, the computer has to:

1. Look in an "index" to see in which "sheet" it has jotted down the current value of  $a$ .
2. Look in the "index" to see in which "sheet" it has jotted down the current value of  $b$ .
3. Copy the two values to a working pad, one on top of the other.
4. Add the two numbers, digit by digit, paying attention to carries.
5. Look in the "index" to see in which "sheet" it keeps the current value of  $c$ .
6. Copy the result to the sheet where it keeps the value of  $c$ .


Yet you are able to write  $c = a + b$  without thinking about the steps above.

We used words like "sheet," "index," and "working pad." We certainly didn't mean to imply that inside the computer there are pieces of paper and mechanical pens! As you well know, the computer is made out of carefully organized pieces of metal, plastic, and silicon. The "sheets," "pads," and "indexes" above are ultimately implemented by trillions of electrons moving about within a complex labyrinth.  It is rather wondrous that millions of such atomic events can be controlled in a fashion that will make them, as a whole, behave much like a

sheet of paper or an index would. It is quite amazing too that this illusion is perpetrated automatically for us, every time we write something as simple as  $c = a + b$ . The illusion is largely successful, but not perfectly so.



© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

### 1.1.1 Execution as Physical Process

While we are developing programs, we work in **an abstract realm**: We think of **variables and data types** rather than of memory chips; of **algorithms** rather than of moving data among those chips; of **program statements** rather than of where and how these statements are stored. Even though we are somewhat aware that variables, algorithms, and statements are fleshed out in a running program in the form of values in memory and CPU commands, we don't usually give much thought to how this happens. We don't give much thought to this because we don't have to: The computer **hardware** and **operating system** are very good at **translating** our abstract programs into electrical currents that change in the appropriate ways. We are so oblivious to that translation that we often forget that it takes place at all, and we confuse the program with its execution, even though the program is a static reflection of thought and the execution is a dynamic process that takes place in the physical world. 

One reason that makes it important to distinguish the program from its execution is that the translation is not quite perfect. Whereas in most cases the running program behaves exactly as we expect, some times the illusion breaks down, and the way in which the computer is physically constructed alters the expected results.

For instance, most programmers don't worry about **running out of memory** when writing programs. Typically, every time you declare a variable in a program you don't question whether there will be enough memory to house that new variable inside your computer. In fact, programming languages rarely provide a comprehensive way of checking for that case, so that even conscientious programmers are often unable to address the possibility under all conditions. For instance, in languages like C++ and Java, each invocation of `new` (to create an instance of a class) can potentially fail due to lack of memory. This is also true of declarations of variable-size objects like strings. This sort of error is also possible in C, even though this language provides us with a model of the computer that is less abstract, more similar to the underlying hardware. Any program may fail even though its logic may be perfectly correct.

 Another effect of the fact that programs execute in the physical world is that even when `a` and `b` are both greater than zero, **`a + b` may be smaller than both `a` and `b`.** Surprised? You should be, if you think that adding data of type `int` is tantamount to adding two integers. We will explain the discrepancy in "[Unit 2. Representation of Data.](#)" 

For the most part, programming languages give us a sublimated view of the computer in which memory is infinite, and in which ints behave like normal integers. These are useful approximations because they hold in most situations, while simplifying very much the task of programming. But they don't hold always! Programs that are correct within that sublimated model of the computer may fail when executed in the real thing—a real computer.

That a program might run out of memory is one of the simpler ways in which the physical nature of execution could affect the way we write programs.

Often, the effects are more subtle. Consider the following code, which is intended to fill an array with integers, starting with zero:

```
#define ARRAY_SIZE 10

void natural_numbers (void) {
    int i;
    int array[ARRAY_SIZE];

    i = 1;

    while (i <= ARRAY_SIZE) {
        array[i] = i - 1;
        i = i + 1;
    }
}
```

When invoked, the procedure `natural_numbers` will never return. That's right, never. But it seems pretty harmless, doesn't it? The loop is certainly not an infinite loop: its index `i` starts at 1 and ends at `ARRAY_SIZE`.

Briefly, the problem is that the compiler allocates the variable `i` to a location in memory that is right at the end of the memory allocated to `array`, so that setting `array[10]` has the unintended effect of setting the value of `i`.

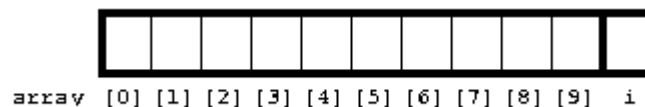


Figure 1: Writing `array[10]` **overwrites** `i`.

We will explore this and other similar problems in "[Unit 3. Memory Layout and Allocation](#)" of this course.

Sometimes, the physical medium in which programs execute does not cause programs to behave "incorrectly," but it causes them to be unexpectedly slow. In other courses, you have learned to **analyze the performance** of a program using **asymptotic analysis**. This technique is very useful when **designing algorithms**, and gives a good approximation to how quickly an algorithm will execute. But like all useful techniques that analyze complex systems, asymptotic analysis takes advantage of simplifying assumptions that don't always hold when the program executes. The limitations of computer technology, for instance, entail, among other things, that memory does not have uniform access times. That is, reading the value of one variable in a program may take much longer than reading the value of another variable. This problem is particularly insidious because we cannot know beforehand which variables will be slower than others, and, worse, the variables that are slow at one point in the program's execution may become fast at a later point. In "[Unit 5. Memory Operation and Performance](#)," we will explore this difficulty, and we will learn how to **speed up programs** that get bogged down by this peculiarity of **real, physical**, computers.

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.1.2 Levels of Abstraction

An "abstraction" is a notion that describes multiple instances of objects or events but that is more succinct than the instances. A good abstraction captures relevant features of the instances while discarding those that are accidental. For example, when we talk about a "car" we may think of an expensive object that moves under its own power and is big enough to carry people, even though there are many types of cars that differ substantially from each other. An electric car is rather different from one powered by a combustion engine, but often we don't care about such differences and prefer to use the **more general concept** of "car." This concept hides the accidental difference in the propulsion system but is nevertheless sufficiently specific to distinguish a car from, say, an airplane.

Abstraction plays a central role in **information technology**. You are already familiar with it. For example, method inheritance in object-oriented languages such as C++ or Java provides a **framework** for abstraction. An "electric car" class can **inherit** methods from the "car" class, which in turn, may inherit methods from a "transportation device" class. Each class captures characteristics that are useful, without specifying characteristics that are not relevant **at its level of abstraction**. Within each of these classes, it is the **implementation** of the public methods that reveals further **details**. The code written for each public method of the "electric car" class may reveal, for instance, that the simple "car" may actually subsume complex objects such as electric generators, batteries, CV joints, and so on. The detail is necessary to design the car, but the abstraction is extremely useful because, once the car is designed, we can effortlessly instantiate thousands of cars to build, say, a model of a city's streets. Imagine what Herculean effort would be required if, in building the city model, we had to separately write the complete code for each part of each car.

Back when computers were first invented this is pretty much what people did: For each computation they needed to carry out, a team of engineers would design a custom computer down to each single electronic tube.

Thousands of such tubes, each very limited, would be interconnected with myriad wires to form a system that would, if not a single wire was out of place, carry out the desired computation. You can imagine how **difficult** and **error-prone** this was. You can also see that each of these engineers had to have comprehensive knowledge of **all** the technology involved, **from algorithms to electronics**.

Things have improved. Today, when you write a small program, you manage to put to work hundreds of millions of transistors, all without needing to know (though you may know) how a transistor works and without needing to actually manipulate any wires. You do this by relying on abstractions.

C++ and Java give you the abstraction of an object space. You describe object classes, and then instantiate objects of those types that behave according to the stipulated **rules**. The compilers for these languages allow you to think in these terms, and free you from having to think at a lower-level, by automatically translating your programs into other programs that

use lower-level abstractions. For instance, some C++ compilers use the simpler notions that are available in C, perhaps implementing a class instance as a C **pointer** that points to a struct which contains pointers for each of the object's **variables** and **methods**, and so on. The resulting C program will behave exactly like the original C++ program, but it will probably be much **longer** and much **harder to read**. Much like cars can be driven by people who don't know how cars work, the C++ compiler allows us to program **using** classes **without knowing** how the classes are **implemented** in a real computer.

But the C programming model is itself an abstraction. The C program that implements the C++ program is itself very simple compared to the actual program that gets executed directly by the CPU. For example, in C, you can name variables with mnemonic names, instead of having to specify cryptically the locations within memory where the variables are stored, and you can **write** `array[i]`, instead of having to compute the location of the *i*th element of array yourself. You can also **write** `c = a + b` instead of having to give the CPU specific instructions on how to carry out the addition. This is because the C compiler takes care of translating C programs into *machine code*. **"Machine code"** is the language that embodies the lower-level abstractions provided by the CPU chip. This is the same for all languages that execute on any one computer, and includes basic instructions like "move byte" and "is a byte zero?"

This is as far as software goes and as far as programmers go. But the engineers at Intel and other hardware companies do not stop there. Machine code instructions are themselves translated by "peripheral" parts of the CPU into even simpler instructions that have meaning deep inside of the CPU. **Deep inside the CPU**, these instructions are, in turn, used to turn switches on and off in a complicated dance achieved by **logic "gates."** These "gates" are in turn implemented with **silicon transistors**, and transistors are themselves abstractions of physical electronic processes.

You may have noticed that abstractions seem to come in **hierarchies**, each level referring to the same underlying facts in more or less detail. A C++ class and a fragment of machine code may both refer to the same program. Whether we choose to manipulate one or the other depends on what we are trying to do. The **higher** levels of abstraction (the C++ class in this case) **lack detail**, whereas the **lower** levels **lack clarity**. Think of a sequence of maps of decreasing scale: the first one is perhaps a plan of your home, and the last one is a map of the world. Now suppose that we made maps of the whole world at all these different scales so that, for instance, your living room would appear in the highest-scale map along with all the other living rooms in the planet. That would be quite a few living rooms, don't you think? Now, suppose that we want to use these maps to find Andorra, a small country that happens to be nestled between Spain and France. The lowest-scale map, the world map that schools hang on their walls, is too coarse to show tiny Andorra. The highest-scale map, however, is far too detailed: it is impossible for us to figure out which living rooms are inside Andorra, and which are in France or Spain—because in such a detailed map, the borders among these countries are no more noticeable than the walls of your house. But suppose that, instead of Andorra, we are looking to find out whether we can fit an extra bookshelf



in our cousin's living room. We would surely be unable to figure this out by looking at a map of Western Europe!

As we have seen, when we program in C++ we work at a pretty **high level of abstraction** that **ignores many details** of how computers work. But there are many **higher levels** as well. A person answering a mobile phone in the street is actually using a computer that is probably partially programmed with C++. Yet for that person, the device is not a computer, and answering the call does not involve thoughts of classes or method invocations.

Why go over this business of abstraction now? Abstraction is ubiquitous, and we know that you hardly need to have it explained. However, while most people readily accept the **advantages** of abstraction, not everyone keeps its **limitations** in mind. For example, the borders drawn between states on a map might gloss over certain details—such as where a house straddles the border between two states—and in that sense, the map might be considered to be **inaccurate** because it implies that entities are either in one state or another. Such inaccuracies do not usually have any practical importance, but, occasionally, they do. It is for such occasions that, while working at a high level of abstraction, we must remain vigilant for cases that show imperfections in the abstractions we use.

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

We have already said that programs written in C, C++, or Java are tricks of magic in the sense that the computer hardware doesn't execute them even though it seems to. Rather, programs are abstract specifications of the less powerful, numerous, and cryptic instructions of machine code that we want the hardware to execute. The hardware doesn't understand classes or variable names directly, it only understands a few data types and carries out only the simplest instructions. If this weren't so, hardware designers would never be able to finish their jobs, overburdened as they are by many of the details of the physical computer with which we are lucky not to be involved.

The illusion that a C program executes on the computer is supported by compilers. A *compiler* is a program that translates other programs from their source language (for example, C) into the language understood by the hardware. Compilers are quite complicated. Consider the following simple loop:

```
for (i = 0; i < 4; i++) {
    sum += array[i];
}
```

The Visual C++ compiler will translate that loop into the following machine code, which is directly executed by the hardware:

110001110100010111110100000000000000000000000000  
11010110000100110001011010001011111100100000111100000  
000000011000100101000101111111001000001101111011111100  
0000010001111101000100101000101101001101111110010001011  
01010101111110000000001100010100100011011101100000100101  
010000100000000010001001010101011111100011101011101111

Isn't it amazing that all those zeros and ones mean something? That they actually mean exactly what the fragment of C code above means? That there is a program that can translate any arbitrary C code into its equivalent sequence of ones and zeros?

One of the many jobs of the compiler is to allocate variables to memory locations, and to keep track of where they are in order to tell the hardware their locations when needed. Each variable in a program needs to be allocated to a specific memory location. The hardware doesn't know the names of variables, it merely operates on the locations that the compiler specifies. For example, in the above machine code, the location of the variable array is given by the number 110110000010010101000010, which we now highlight in the original machine code:

[illegible]

Note that from the machine code, it is nearly impossible to tell whether that sequence of zeros and ones is a variable. Even if we undertook a thorough analysis of the machine code, armed with detailed knowledge of the CPU architecture, we might at most be able to determine that the sequence of digits is the location of a value that gets added to something, but in no case would we be able to derive that the original name of that value was array. So we see that machine code is not only hard to interpret, but it also contains incomplete information about the original source code. In turn, it contains more specifics about how exactly that source code will be executed by the particular CPU in this particular computer. This is perhaps not unlike a detailed topographic map of a forest that, containing detailed information about individual hills, fails to tell us how to get to the forest from a distant city.

In practice, there are not, as this example suggests, just two levels of abstraction—**source code** and **binary machine code**. There can be one or many intermediate levels, each designed to help understand and communicate at some level of detail. For example, you may have noticed that after you compile a program, but before you run it, a step called **"linking"** is performed. Linking combines a program with various **system and language code libraries** to implement built-in system and language functions. The linker program would not be able to do its job with only binary information as shown above. On the other hand, the linker has no need for the original source code. The linker deals with an intermediate program representation called an object file, which contains binary instructions and information about where certain "external" functions and variables are referenced.

Now let's turn our attention to "debugging." Debugging is the task of getting "bugs" out of a program. A "bug" is simply an **error**. Errors in programs receive this **colorful name** because one of the first errors in one of the earliest computers turned out to come from a short circuit caused by a **real insect** that had been trapped in the computer's circuitry.

How would you go about debugging a complex program? In complex programs, an error often does not become evident immediately. The unexpected condition introduced by the programming error may, meanwhile, affect many other parts of the program that are without fault. When faced with an erroneous symptom, then, we are often confronted with many things that seem wrong, and we cannot immediately tell which is the cause we ought to fix and which are effects we should ignore. The art of debugging is hard detective work, illuminated alternatively by **perseverance**, **strict application of logic**, **inspiration**, and **experience**.

Like detective work, debugging requires the testing of **hypotheses**. In debugging, a hypothesis might propose that a particular chain of events

occurred. Testing that hypothesis then requires looking at the events that took place during the program's execution.

Programs execute in steps. Each step modifies a small number of variables. Looking at the history of execution is equivalent to executing the program anew, under the same conditions that caused the problem to appear, but stopping the execution at selected points before the problem actually appeared. An inspection of the suspect variables at those points will either confirm or deny the hypothesis. The tool that allows us to stop the program and inspect its variables in the middle of its execution is called a *debugger*.

Let's now think about how a debugger works. Remember that what the computer executes is not the source code we wrote, but the machine code that the compiler generated. The first difficulty the debugger faces is to understand which sequence of zeros and ones corresponds to the program statement where we wish to stop execution. The next difficulty is that to tell us the value of a variable (for example, *array* in the example above), the debugger needs to find out the location to which the compiler assigned the variable whose value we desire to inspect. As you may imagine, this is quite complicated. Debuggers need to relate machine code to the original source code, and they do this by reversing the translation that the compiler carried out when generating machine code from source code.

As shown above, pure binary machine code is pretty opaque, even to a debugger, so it cannot really reverse the compilation process without help. When the compiler generates machine code, it can also generate information to assist the debugger. This information tells the debugger where to find variables in memory and what instructions correspond to what lines of source code.

If you were to dismantle a car engine, you would find yourself surrounded with thousands of nuts and bolts, all of which look the same. Armed with patience and foresight, you could label each nut and bolt as you disassemble the engine. For example, you could write "carburetor jet needle" on a certain small metal cylinder (which would end up getting lost anyway). But in actual practice, of course, nobody does it that way. People just poke around under the hood, in a sea of parts, and try to be alert enough to keep track of everything in their heads. This is also how people used to look "under the hood" of computer programs, when computer programs were not much more complicated than a car engine. Today, however, programs are very much more complicated than mere mechanical devices, and a more systematic approach is needed. Debuggers allow us to poke under the hood of computer programs by allowing us to label each byte with its higher function so that we don't have to keep track of what every raw byte means. In the rest of this unit, we will see how debuggers help us in this way.

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.1.4 The Visual C++ Environment

We have seen that the task of developing programs involves, to no small extent, the repeated translation of programs from the source level to the machine level, and vice-versa. Compilers translate source code into machine code, and debuggers do something of a reverse translation. Programming, of course, also involves executing machine code as well as managing source code.

The management of source code is no simple matter: large programs are divided among many files that are interdependent and that use many pieces provided by third parties and the operating system. To compile the source files of such large programs in the right order is not as easy as it might sound! The process of software development can take advantage of many types of tools, beyond the necessary compilers and debuggers.

Traditionally, software development tools came packaged separately. They might all have come from the same vendor, but each of them worked independently of the others. Of course, the different tools needed to share some information. For example, you may remember from the discussion in page "[1.1.3 Compilers and Debuggers](#)" that the debugger requires the cooperation of the compiler in order to be able to refer to memory locations with the names given in the source code rather than in the cryptic numbers of the machine code. In traditional development environments, information was shared among the separate tools through the use of files that had a standard, agreed-upon format.

Several years ago software companies started to sell "integrated development environments." These are large programs that encompass, within a single package, all the tools needed for developing software. The goal was to reduce the time it took to go through the write-compile-debug cycle by allowing tools to communicate with each other with greater agility. For example, an

integrated environment is able to do “**incremental compilation**,” in which the program is compiled in pieces—even as we write other pieces of the same program. Visual C++ is one such integrated environment.

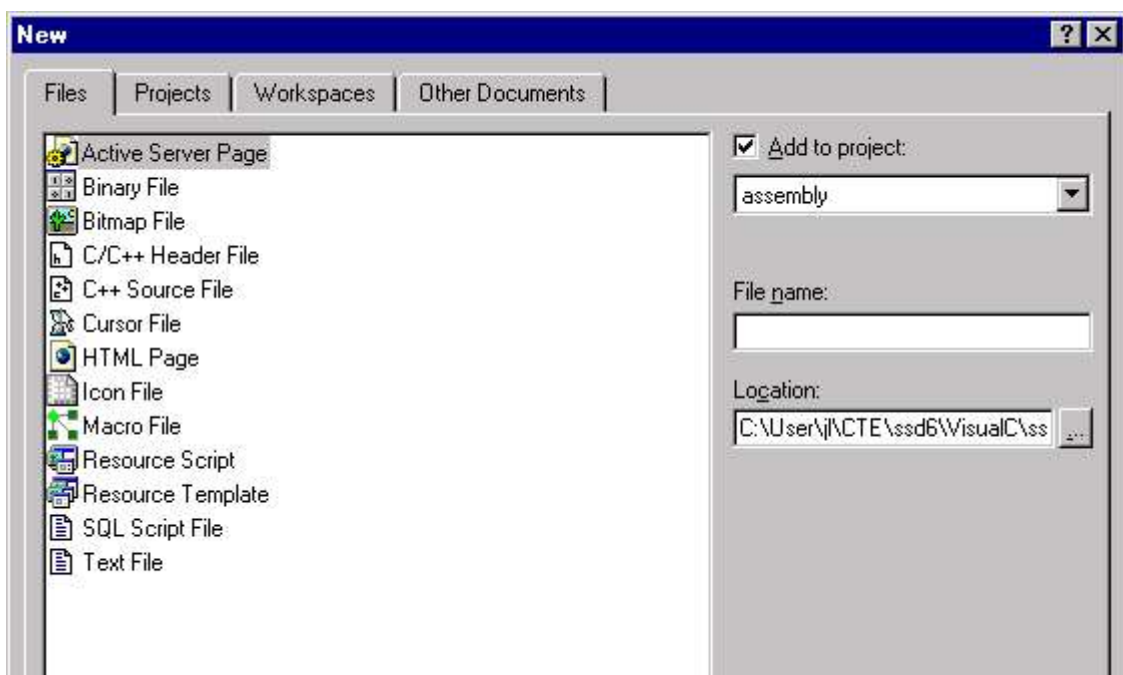
While integrated environments offer clear advantages, they have one **disadvantage**: we must buy into the environment as a whole. We cannot, as we could with separate tools, mix and match tools of different strengths and from different vendors. For instance, if you use Visual C++ you are almost forced to use the Visual C++ **editor**, even though you might prefer another one. (You can use other editors for routine programming, but external editors do not provide all of the features and integration of the built-in one.) Even though this is not such a terrible thing, some developers avoid integrated environments for this reason.

Visual C++ is a very large utility and has many components that we will not be using in this course. We invite you to browse its **documentation** whenever curiosity strikes you, but do not feel uncomfortable if you come across menu items you do not understand. Much of the size of Visual C++ is devoted to what Microsoft calls “**application wizards**.” Application wizards are tools that write programs for you. There are certain types of programs that people need to write very frequently and that are very similar to each other. An application wizard will write most of these for the programmer, who can then customize them with only a little effort. For example, using Visual C++, you can create an application that is identical to the “Notepad” application without ever writing a single line of code. In this course, we will not use any application wizards, but you should be aware they exist: you might find a use for them later in your career. We don’t want you to be confused by the myriad of choices that some of the Visual C++ dialogs will offer you. Despite its complexity, Visual C++ is not

hard to use if you restrict yourself to its basic functionality.

Visual C++ has three notions that we will need to understand: **workspaces**, **projects**, and **files**. A project is a collection of files and the rules that define their interdependencies. A workspace is a collection of projects that are stored in the same region of your disk. While **files** within a project are intimately **linked** to each other, projects within a workspace are not. In order to compile all the files of a project into an executable program, Visual C++ needs to know how they **depend** on each other, and in what **order** to compile them. While **projects** within a workspace might also be related, they can be **compiled separately**. For example, one project might be a library (DLL) that is used by another project but compiled independently.

Projects may contain files of several types. You will see that the "new" dialog has a "files" tab that lists the possible file types (see the figure of this dialog below). An application needs to have icons, figures, and documentation—besides the source code that makes it run. There is a file type for each of these needs. In this course, we will restrict ourselves to files of type "C/C++ Source File" and "C/C++ Header File."





One last remark: Visual C++ is able to compile and debug programs written in both C and C++. C++ was designed to be compatible with C so C programs can be compiled with a C++ compiler without notice or preparation. In this course we will be programming in C, but we will be using Visual C++ for the purpose of compiling and debugging the programs we write.

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.



## 1.2 The Visual C++ Debugger

This module is eminently **practical**. We will show you how to **create** a program in Visual C++. We will also show you how to use the **debugger**. The modules that follow will use the debugger to show several things about the programs you write.

- 1.2.1 [Creating a C Program Using Visual C++](#)
- 1.2.2 [Breakpoints and Steps](#)
- 1.2.3 [Examining Data](#)
- 1.2.4 [Example](#)

### Assessments

- [Multiple-Choice Quiz \(2\)](#)

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.2.1 Creating a C Program Using Visual C++

We now describe how to create an empty Visual C++ workspace. We then create an empty project within that workspace, and, finally, we create a C source program within that empty project. As we have discussed in "[1.1.4 The Visual C++ Environment](#)," in doing this we will be ignoring many of the Visual C++ features that will be presented. As we describe the steps, we will expect that you will be carrying them out in your computer's copy of Visual C++.

Start Visual C++. You are likely to find it under the **Start** menu, within the **Programs** and **Microsoft Visual Studio** submenus. The figure shows how the **Start** menu selection would look in our computer: in yours it will be a little different.

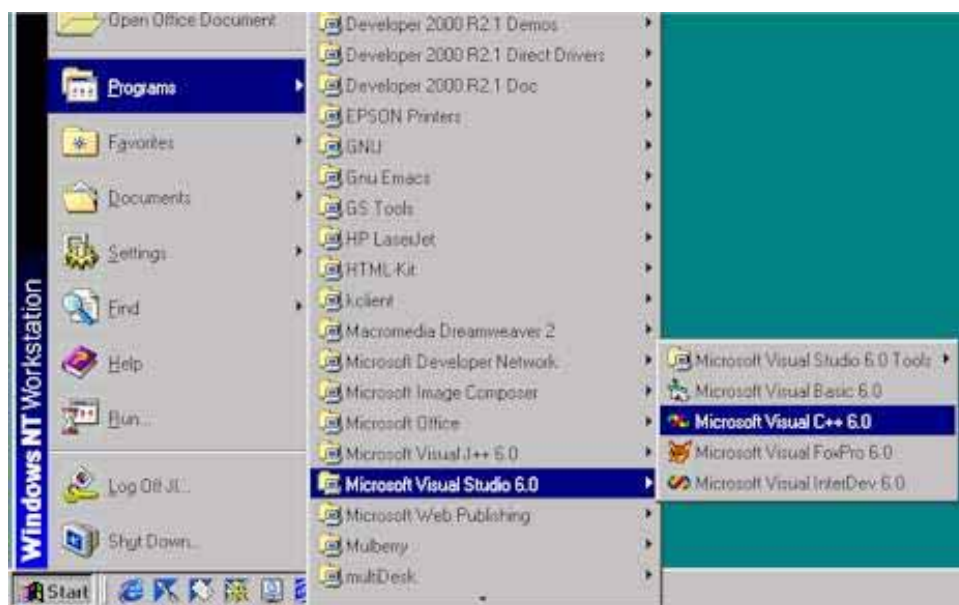
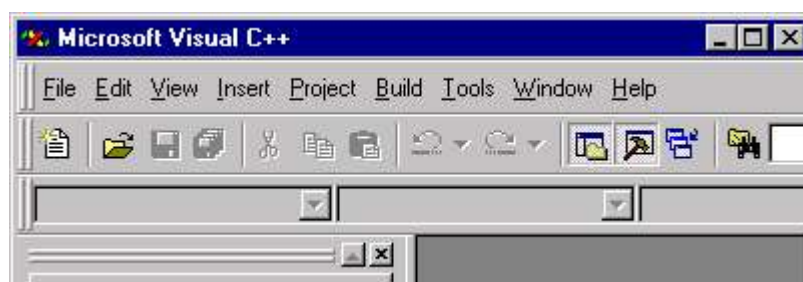


Figure 1: How to start Visual C++

Once Visual C++ starts, you will see a window that will look like the one in the figure below. Besides the usual **menu and tool bars**, you can see the window is divided into three panes. The lower pane is the **status area**, where the various tools of Visual C++ (for example, compiler, debugger, text search) report their progress and errors. You can see there is a tab for each tool. Above that, the pane on the left is the **project area**, where the various workspaces, projects, and files will be shown once we create them. Finally, the pane on the right is where we will see the **source files** when we start editing them.



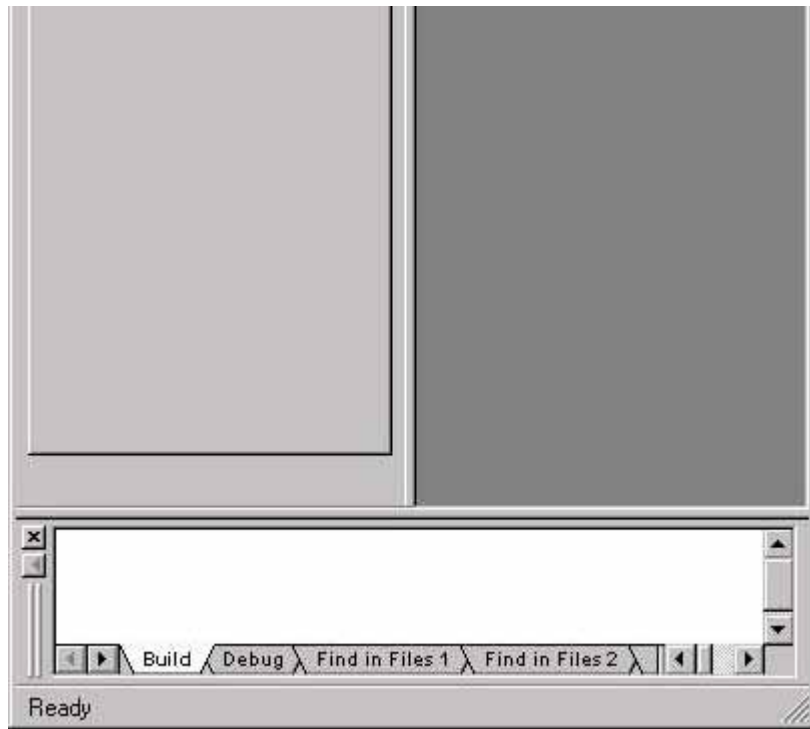


Figure 2: First Visual C++ Screen

We are now ready to create an empty workspace. Pull down the **File** menu, and choose **New...** You will get a new window with several tabs. Click on the **Workspaces** tab. You should see something like the picture below. Click on **Blank Workspace**, and select a name for your new workspace. As the picture shows, we have named ours "CTE SSD6," but you may name yours anything you like.

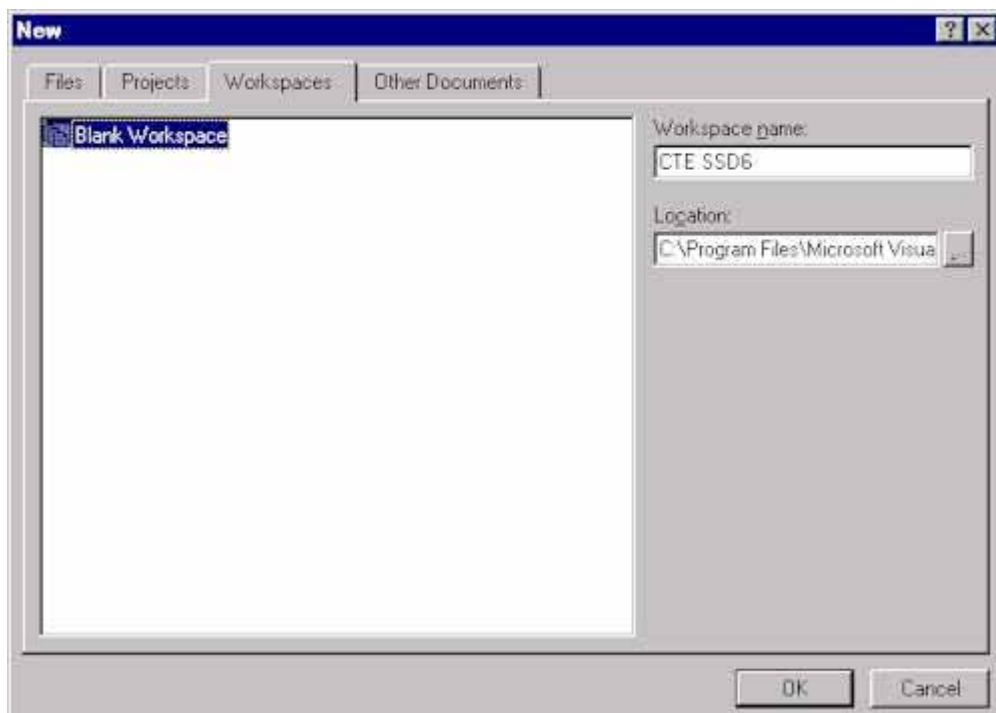


Figure 3: New Workspace window

Now you will see that, in the project pane, you have a new workspace.

Next, we create an empty project within this workspace. Click once on the workspace, and select again the **New...** item from the **File** pull-down menu. This time, select the **Projects** tab. The window should look similar to the one below. Click once on the **Win32 Console Application** icon, select the **Add to current workspace** radio button, and type in a name for your new project. We have called our project "Hello World."

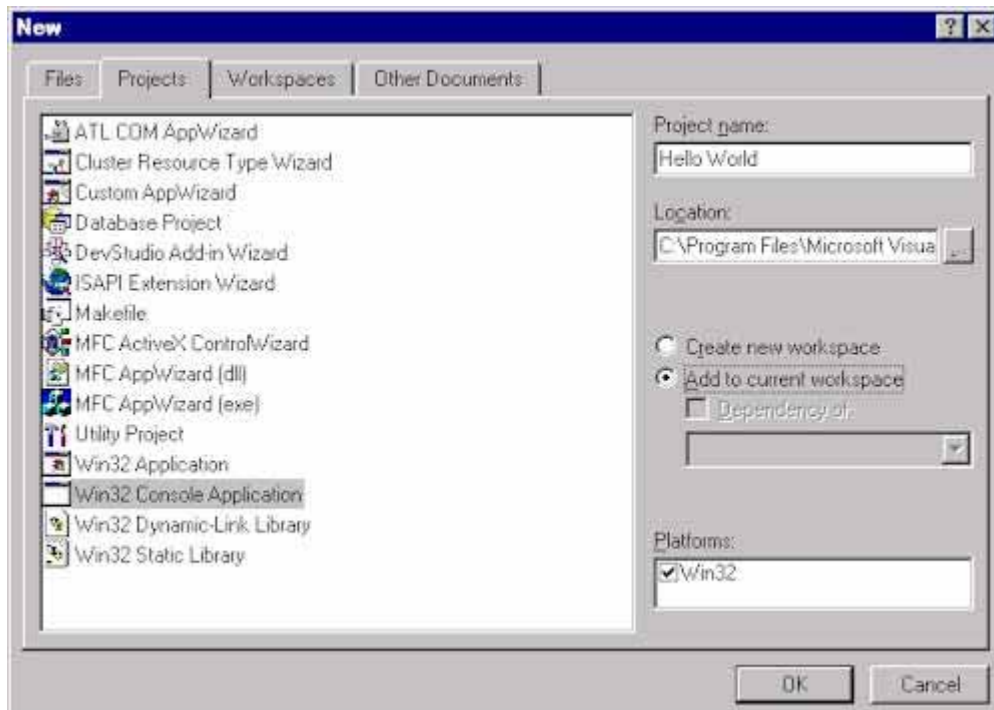


Figure 4: New Project Window

We asked you to select **Win32 Console Application** out of the many choices available. Each of the choices here selects one from among many project types. The main difference between project types is which application wizard, if any, Visual C++ will apply to your project. For instance, if you had selected **Database Project**, Visual C++ would do you the favor of creating a template program that follows the pattern of most database interface applications, which you would only need to customize rather than write from scratch. **Win32 Console Application** is the simplest project type available.

When you click **OK**, you get another window like the one below that will ask you which of several subtypes of projects you would like. Here, Visual C++ is basically disbelieving that you want such a simple project type, and is trying to convince you to accept some of its help. It really wants to write code for you. But along with Visual C++'s help comes a host of issues and design decisions that we don't want in this course. Select **An empty project**, as the figure shows, and click **Finish**. You will get an information window telling you, basically, that Visual C++ has done nothing for you. Had you chosen another project type, this window would contain a list of the files that Visual C++ had created on your behalf. Click **OK** to make it go away.

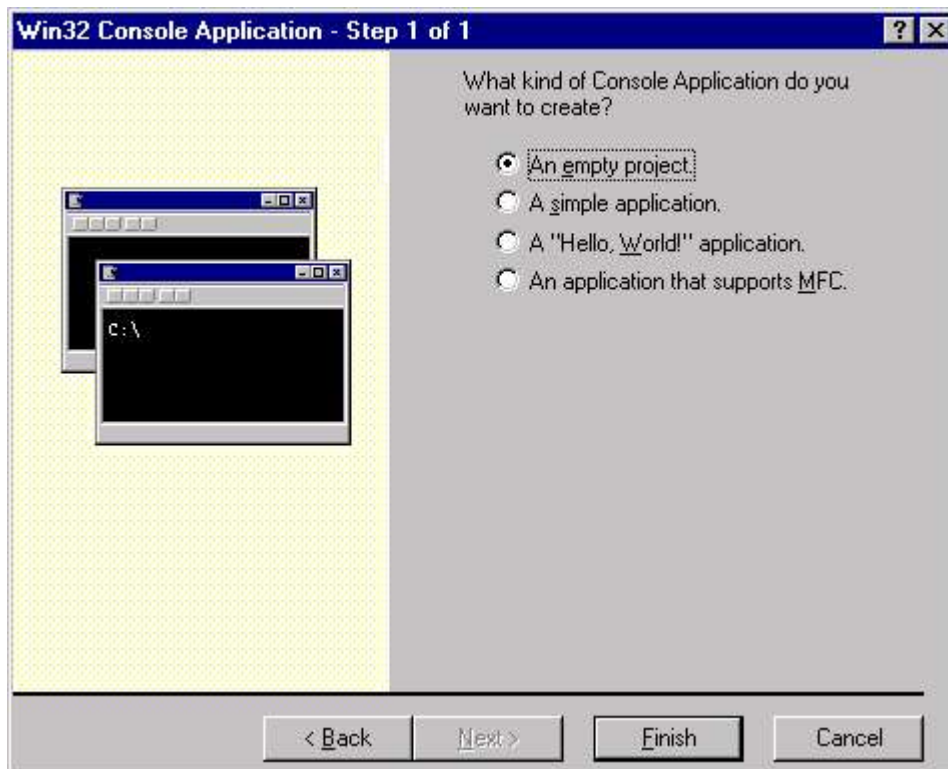


Figure 5: Selection of Console Project Type

Back in the main Visual C++ window, within the project pane and within your new workspace, you now have the new project. We only need to create some files for this project. Select **New...** once more, and select the **Files** tab—if it is not automatically selected. The window will look similar to the one below. Select **C++ Source File** from the options, and type in the name of the new file. We have named the file "main," but you can name it anything you want. Click **OK**. Back in the main Visual C++ window, you will see that the new file (though empty) now shows up inside the editor. We are ready to start typing a program into it!

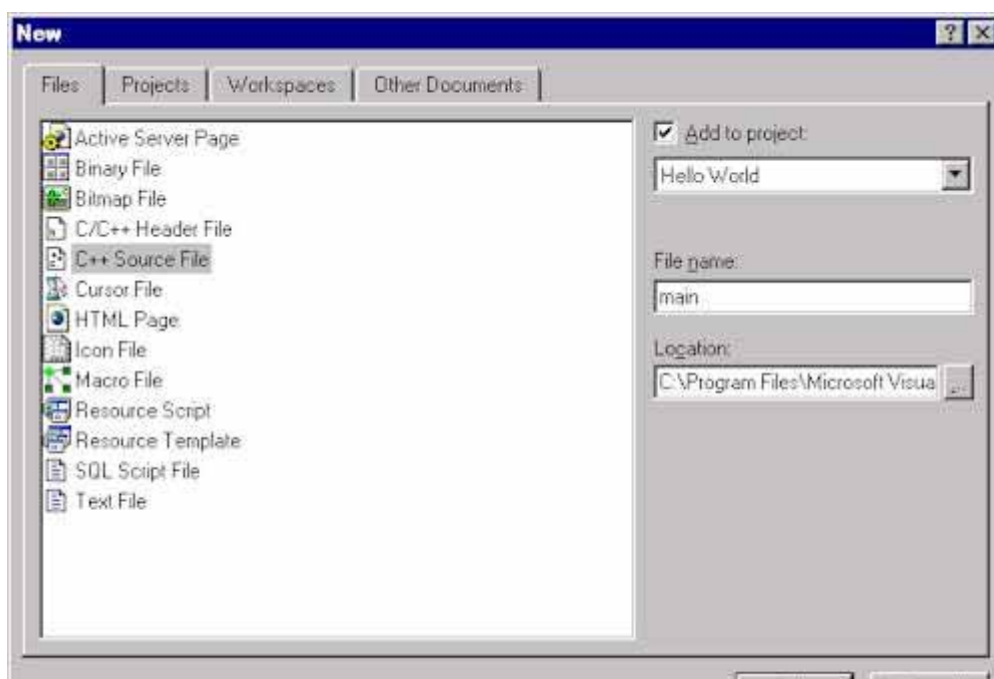




Figure 6: New File Window

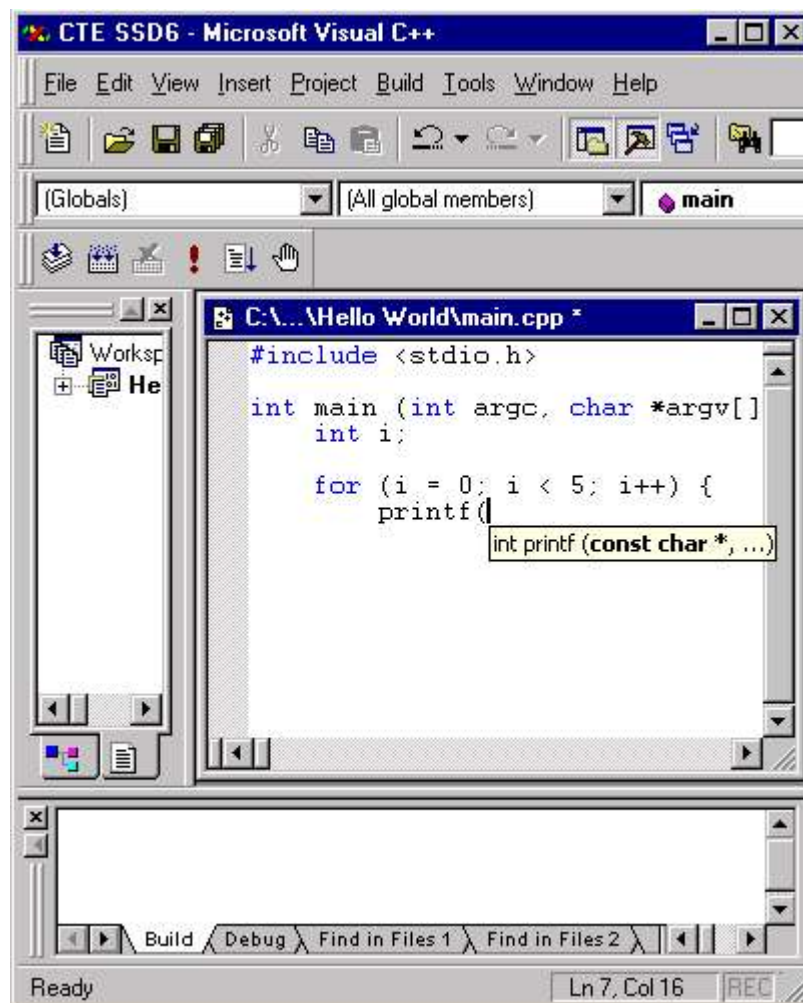
Type the following into the edit window:

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i;

    for (i = 0; i < 5; i++) {
        printf(
```

Note two things:

- As you type, words that have meaning in C, like `int` and `for`, **change color**.
- When you type the parenthesis right after `printf`, a small window appears. `printf` is a standard C procedure, that is to say, one that comes with the language when you include the header file `stdio.h`. So you don't have to write it, and Visual C++ already knows about it. Visual C++ tries to help you remember what **parameters** this procedure takes by showing you **the prototype of the procedure** within the small window.



File Edit View Insert Project Build Tools Window Help

figure 1: typing code

Now, finish writing the program by typing:

```
        "Hello World\n");  
    }  
    return 0;  
}
```

You have just completed the program. Compile it by selecting **Compile main.cpp** from the **Build** pull-down menu. The compiler will tell you what it does through the status pane at the bottom of the window. You are not yet ready to run the program. What you have done is to **generate machine code for the source code you wrote.** But your code uses `printf`, which you did not write. You still need to **"link"** your machine code with the machine code provided by the system that implements `printf`. To link everything into an executable program, select **Build Hello World.exe** from the **Build** pull-down menu. Now you have a program that you can execute, from within or from without Visual C++. Try running it from within Visual C++ by selecting **Execute...** from the **Build** pull-down menu. Presto!

Before you exit Visual C++, you should save everything.

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.




## 1.2.2 Breakpoints and Steps

We will now start to use the Visual C++ debugger. We will apply it to the program that we have created. The program is the following:

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i;

    for (i = 0; i < 5; i++) {
        printf("Hello World\n");
    }
    return 0;
}
```

Within Visual C++, open the workspace by selecting **Open Workspace...** from the **File** pull-down menu. You will need to click on the file of extension **.dsw**. You should have gotten your program back.

When you start the debugger, Visual C++ automatically starts running your program. Do this now, by selecting **Go** from within **Start Debug** in the **Build** pull-down menu. You can also achieve the same effect by pressing **F5** or by clicking on the  icon on the toolbar.

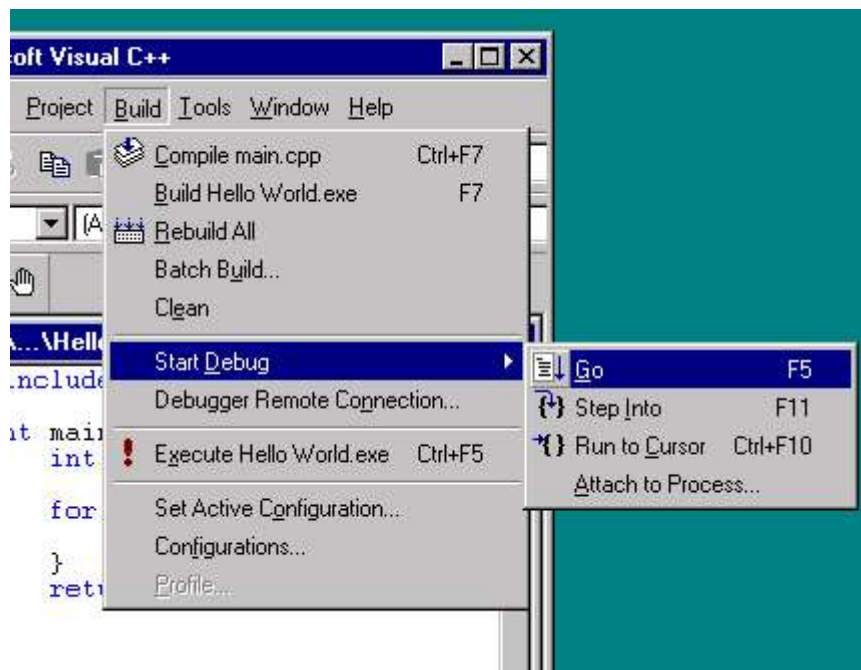



Figure 1: Starting the debugger

When you do so, you should see the console window quickly appear and disappear. We bet you did not expect this. What happened is that the debugger started, and finished executing your program.

For the debugger to be useful, we need to set **breakpoints**. A breakpoint,



as its name implies, is a point in the program at which execution will be stopped (or "broken"). Once execution stops, you can use the debugger to snoop around, inspect variable values, and observe the general state of the computer. Let's try to insert a breakpoint in the line that calls `printf`. While the mouse pointer is over that line, press the right button of the mouse. A menu will pop up. Select **Insert/Remove Breakpoint**, and click again. You can achieve the same effect by clicking on the `printf` line and then clicking on the  icon on the toolbar.

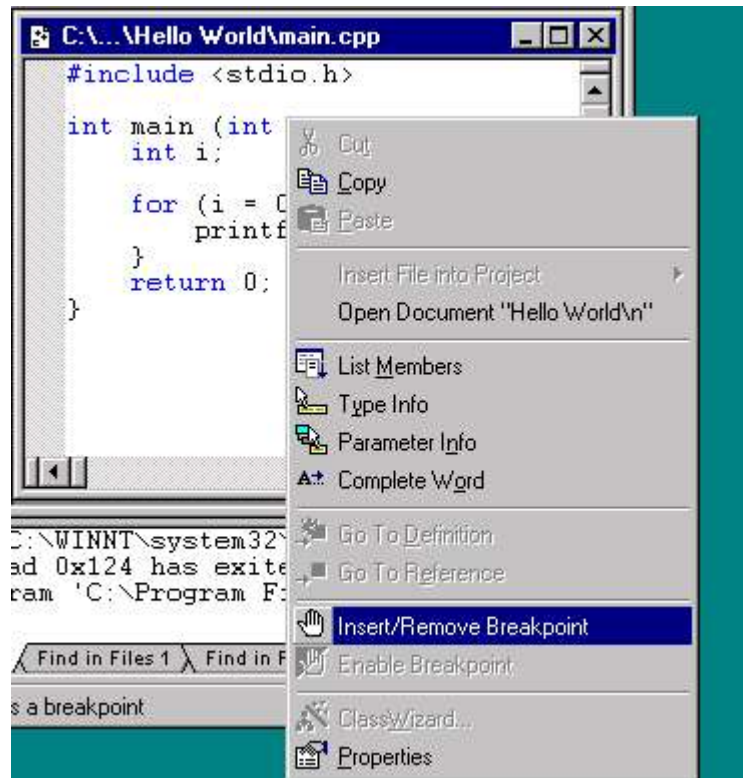




Figure 2: Inserting a breakpoint

A **red circle** will have appeared to the left of the line. You can think of this red circle as a stop sign. You may insert as many breakpoints as you want. The debugger will **stop execution** at each of them. Let's see what happens now if you start the debugger. **Press F5 again.** You will see that this time, the console window stays around, empty, and that a yellow arrow appears near the "stop sign." This arrow indicates that the program is currently executing at that point.

From within the debugger, we can control execution in several ways. Let's try two of these. In the debug toolbar, you see the following two icons:

-  Will continue execution, and automatically stop it at the beginning of **the next line** of source code. (This can also be done by selecting **Step Over** in the **Debug** pull-down menu.)
-  Will continue execution until the program **reaches the current location**, or until it **reaches a breakpoint**, whichever comes first. (This can also be done by selecting **Run to Cursor** in the **Debug** pull-down menu.)

Try clicking on the **Step Over** button. You will see that the yellow arrow moved to the next line, indicating that the execution is now stopped at the end of the loop. You can see that the program did execute the printf because a single "Hello World" now appears in the console window. Now, try clicking the **Run to Cursor** button. The yellow arrow is back on the stop sign, indicating that the program's execution continued, coming around the loop, and not stopping until the breakpoint was again reached. If you click on these buttons a couple of times, you will see that the execution goes around the loop, stopping in every iteration, and printing "Hello World" one at a time.

Let's now delete the breakpoint and insert another one in the line "return 0;". To delete the breakpoint, place the pointer over the stop sign and right-click the mouse. Select **Remove Breakpoint** from the menu. The stop sign is gone. Now, insert a breakpoint in the "return 0;" line. Things will look like this:

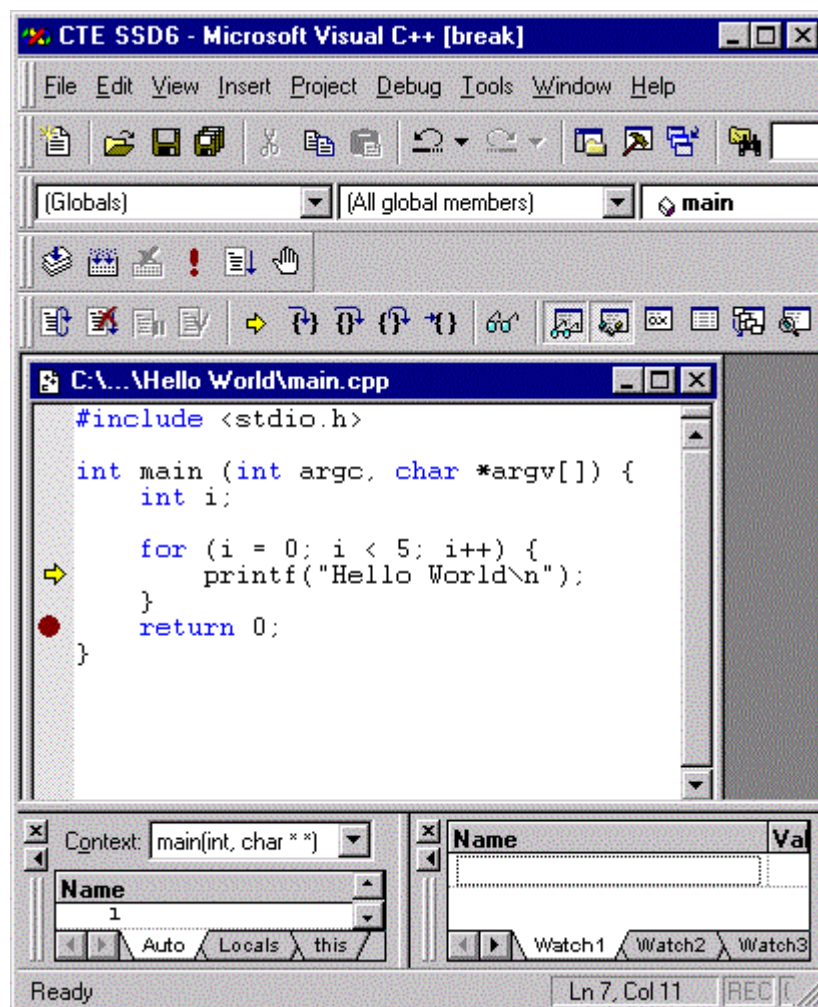


Figure 3: After inserting the second breakpoint

Click on **Run to Cursor**. The execution will again stop at the printf, even though there is no longer a breakpoint there. That is because we clicked on **Run to Cursor** when the yellow arrow was in that line.

To continue executing the program until the next breakpoint, which is now

after the loop, we need either to press F5 or to select **Go** from the **Debug** pull-down menu. Try it.

There are two other ways in which we can control program execution, ways that we will not be showing. **The Step Into button is equivalent to Step Over—unless the program is stopped at a line that has a procedure call.** In that case, **Step Into** will stop in the first line of the callee. The **Step Out** button continues execution until the current function returns, and then stops it at the caller.

There are also many other ways of setting breakpoints. For instance, we can specify that execution does not stop until the fourth time the program happens upon a breakpoint. Or we can specify that the program does not stop unless a particular **condition** (for example, `i == 4`) is true. To modify breakpoints in this manner, navigate around the **Breakpoints...** dialog of the **Edit** pull-down menu. Henceforth, we won't be covering the various features of Visual C++ in detail: we will assume that you are sufficiently familiar with it to strike out on your own—aided by courage and the comprehensive Visual C++ help facility.

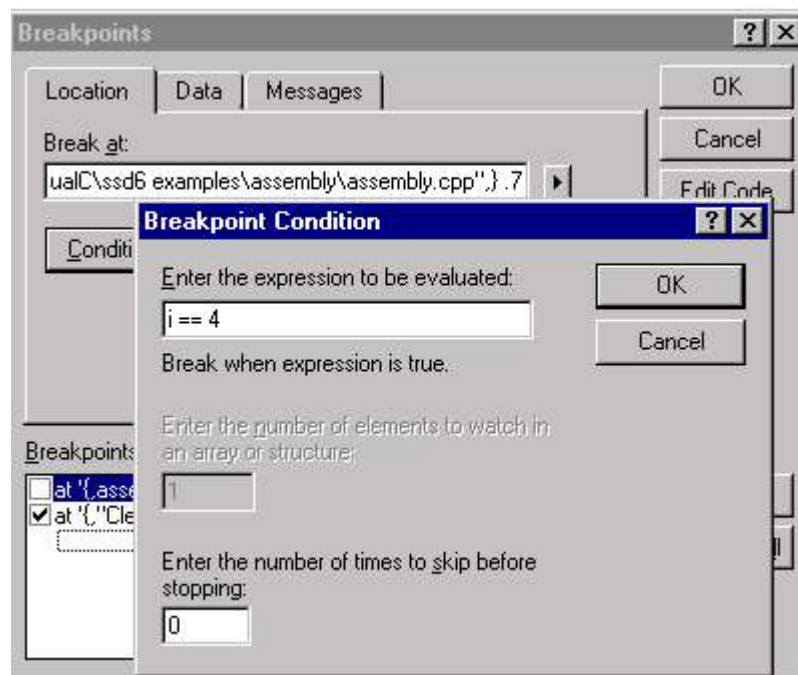



Figure 4: Edit Conditional Breakpoint Dialog

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.2.3 Examining Data


There are three main ways to **inspect the values of variables** in Visual C++:

- While debugging a program, the value of a variable appears in a small window when the **mouse pointer** pauses briefly over the name of that variable in the source code window.
- While debugging, **a pane below** the source code window shows the values of selected variables. There are two useful selections: **the Auto selection and the Locals selection**. The **Auto** tab shows variables that are automatically deemed of interest at the point where the program is currently executing. These may or may not be the variables that you are interested in; Visual C++ guesses what you might be interested in, but it cannot be all knowing. The "Locals" tab shows all variables that are local to the currently executing function, including the parameters to the function.
- By clicking on the  icon, a second pane opens below the source code window. This is **the Watch pane**. In here we can type any valid C expression whose value we are interested in. The expression is evaluated in the scope of the current breakpoint. These expressions can be simple variables, or any other expression that could appear on the right-hand side of a C assignment.

You can also modify the current value of a variable *while* the program is running, by displaying the value using the **Variable** or **Watch** windows, or both, and double-clicking on the value.

Let's **practice**. Open the workspace where you saved our "Hello World" program, and insert a breakpoint in the `printf` line. Then start up the debugger (F5) and position the mouse pointer over one of the appearances of `i` in the `for` line. You will see that the value of `i` appears. Press F5 again to let the debugger go through one iteration, and again place the mousepointer over `i`. Has the value changed? Convince yourself that you understand what is going on.

Now pay attention to the variable pane below. It should show the value for `i`. It may show in black or in red. Red signifies that that value has changed since the program was last stopped. This meaning of "red" is consistent across Visual C++, as we will see. But be warned: Visual C++ is not always good at keeping track of changes when we open and close variable and watch panes. You should not trust the color of the values absolutely; always keep track yourself, when in doubt. Press F5 again, just to see the value of `i` in the variable pane change.

Now, click on the  icon until a second pane appears to the right of the variable pane. This new pane will be empty. Click on it, type `"i"`, then Enter. You will see that the value of `i` again appears and behaves like the entry in the variable pane. Repeat the process, but this time type `"i == 4"`, and then again type `"i + 2"`. You will see that the values of these expressions at the current point of execution appear and are recomputed

every time the program hits a breakpoint.

Try changing the value of `i`, for example, to 0 and continue running the program. How many of the "Hello World" do you get now? Do you understand why?

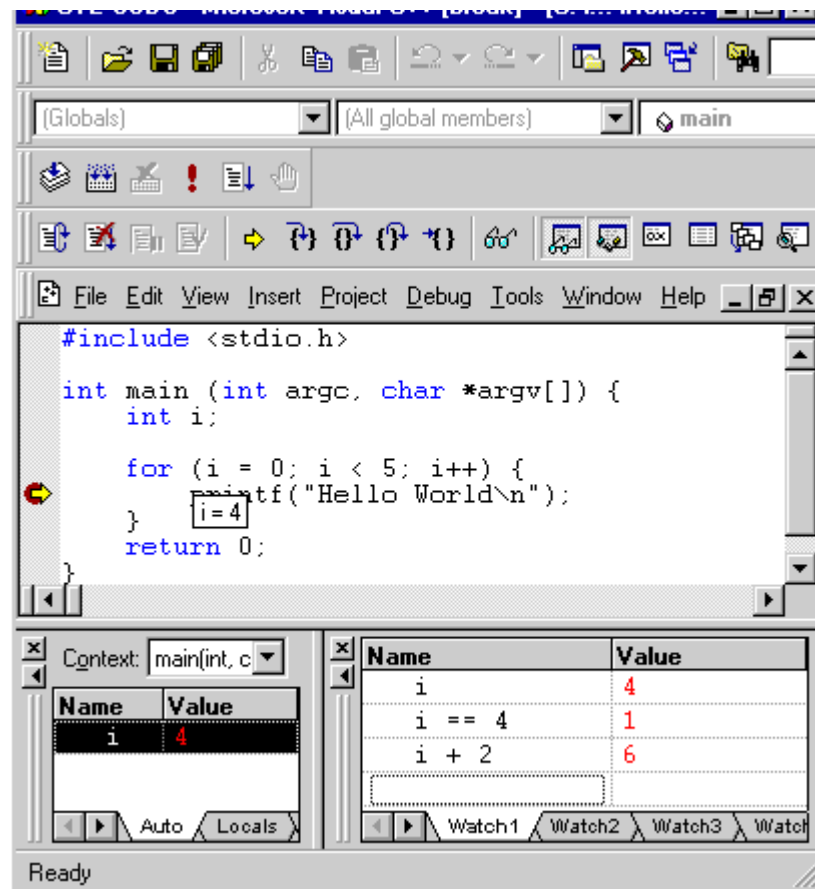


Figure 1: Examining data using the Watch pane

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.2.4 Example

Time to **debug!** The program below has a problem. It is intended to take anything we type into it and output it back. But it doesn't quite do that. For example, if we type in "Hello", it does output "Hello". If we then type in "Hello World", it correctly outputs "Hello World". However, if we again type "Hello", it insists on printing "Hello World". It's really enamored with the "World" word, and won't drop it! For your playing pleasure, the program is also available in the file [Echo.cpp](#).

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

#define MAXLINE_LENGTH 80

char Buffer[MAXLINE_LENGTH];

char * readString(void)
{
    int nextInChar;
    int nextLocation;

    printf("Input> ");
    nextLocation = 0;
    while ((nextInChar = getchar()) != '\n' &&
           nextInChar != EOF) {
        Buffer[nextLocation++] = nextInChar;
    }
    return Buffer;
}

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

    do {
        newString = readString();
        printf("%s\n", newString);
    } while (strcmp(newString, "exit", 4));
    return 0;
}
```

First we ought to understand the code. To do so, you have to be a little familiar with C. This shouldn't be a problem, since you are already likely to be familiar with C++, and the two are fairly similar. However, you may need to review a thing or two and this will be a good opportunity. Next, we need to understand what some of the library functions do. The program calls the functions `strcmp()` and `getchar()`. The documentation for all library functions is available within the Visual C++ help facility. An easy way of getting the relevant documentation is to highlight the name of the function you want to know more about, and then press F1. You can also get to the documentation by starting from the **Help** pull-down menu. Check out the documentation, and examine the program above until you understand what it tries to do and how.



Do you understand the program now? Okay, here is an example of what it does when we type the instances of "Hello" as described above:

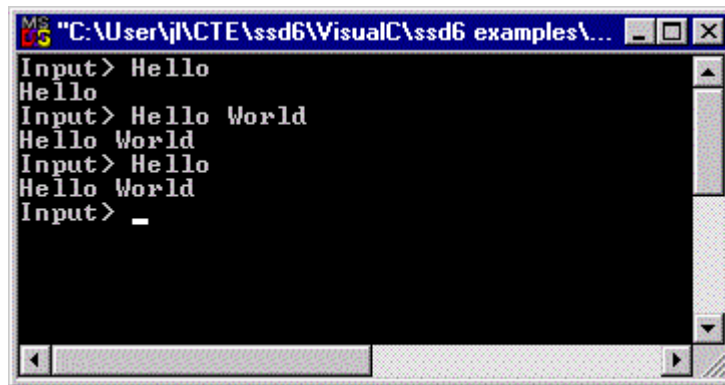


Figure 1: Behavior of the Echo program

Try it yourself. [Download](#) the file, make a new Visual C++ project, and run it.

Programmers often debug by working backwards from **the known erroneous symptoms**. That is what we will do here. The symptom is that when we run through the loop in `main()` the third time (when we give the program input as in Figure 1), the `printf` prints the wrong string. Let's first make sure that **the value of the string** is indeed wrong at that point: if it weren't, we would have to blame **`printf()`** for the problem, unlikely as that might be.

Insert a breakpoint at the `printf()` line, then press F5, and type "Hello" into the console window. Press F5 again, and type "Hello World". Once more, press F5 and type "Hello". You will have just taken the program to the brink, to the point immediately before the bug manifests itself. Take a look at the value of `newString`. It should be "Hello", but it is "Hello World" instead. So the problem is not within `printf` after all. We have discarded this one possibility.

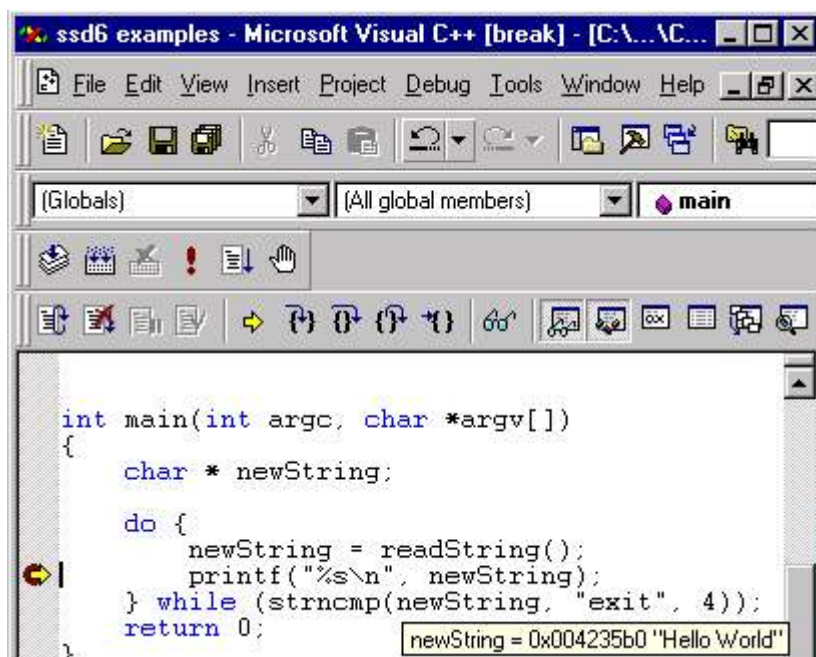




Figure 2: Printf is OK

If the value is already wrong when `printf` is called, then it has to be wrong when `readString` returns. Let's examine the way `readString` constructs `newString` by placing a breakpoint within it. We would like to insert the breakpoint inside the while loop, but think of what would happen if we did that. How many times would we have to press F5 to get the program to the point where our problem arises? We would have to do this once for every character that we typed in up to that point. That's 18 times, including the new line at the end of each input.

Instead, we can insert a breakpoint before the while loop, let it go twice, and *then* set the breakpoint inside the loop. This way, we only need to press F5 three times. Let's do this, and then let's take a look at the value of `Buffer`. At this point, the program hasn't yet started to fill in the string that `readString` will return, but `Buffer` already has the value "Hello World"! How could this happen?

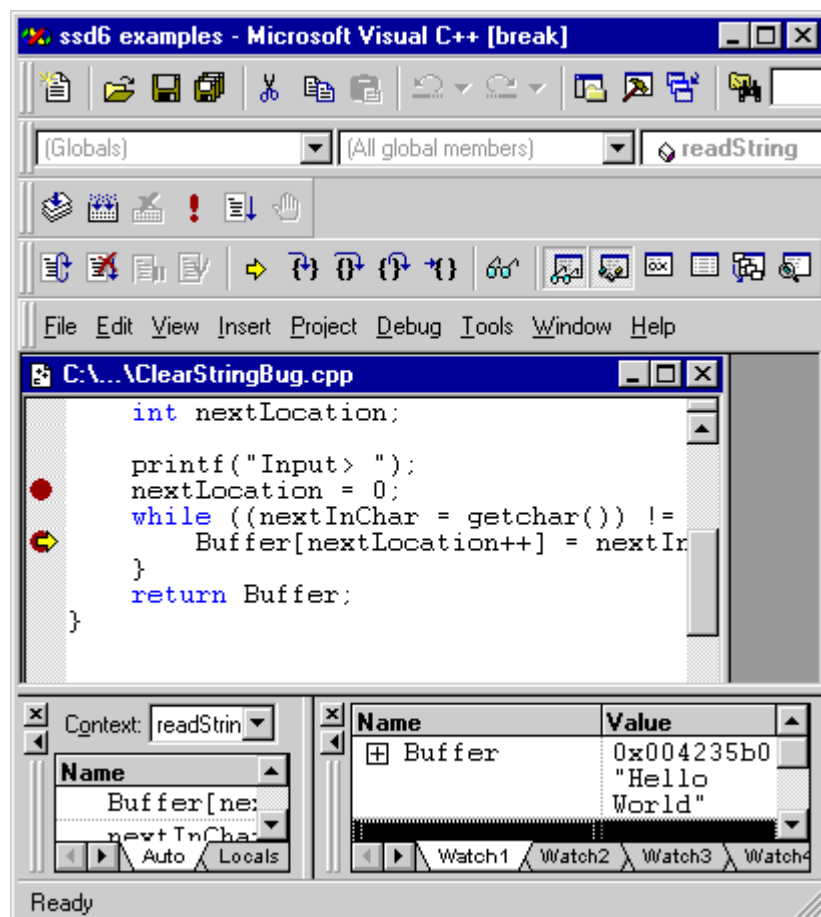




Figure 3: "Hello World" is already there!

This happened because, in C, there is no string data type. Strings in C are no more than arrays of characters. It is a convention that strings are null-terminated, that is, that the string contained within a character array ends with the first zero (`"\0"`). But this is a convention that is not enforced by mere usage of the character arrays. The zeros that terminate strings have to be inserted explicitly by programs that manipulate them. In this case, the array `Buffer` still contains the value that it had at the end of the previous call to `readString`. Whereas the current call will overwrite the first five characters in the array with `"H"`, `"e"`, `"l"`, `"l"`, and `"o"`—the remaining six characters from the previous call will remain as part of the character array that `readString` will return this time.

How would you fix this?

C and C++ are similar, but as you have just seen, C has fewer built-in types and other abstract machinery than C++ does. For this reason, many people prefer to program in C++. But C's relative **lack of abstraction is also its strength**. Because C programs are more similar to the way the computer actually executes programs, C affords greater control and insight during execution.

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.

## 1.3 Variables and Addresses

- 1.3.1 [Addresses and Naming](#)
- 1.3.2 [Pointers](#)
- 1.3.3 [Examining Memory](#)
- 1.3.4 [Arrays and Strings](#)
- 1.3.5 [Naughty Pointers](#)

### Assessments

- [Multiple-Choice Quiz \(3\)](#)

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.3.1 Addresses and Naming

In hardware, all data is stored in memory. All of a program's **variables** and **code** can be found in memory. Memory is a sequence of bytes that is numbered, starting with 0. The machine code that the CPU executes operates on memory locations, identified only by their number. These numbers are called **"addresses."** **Compilers** take care of translating the operations our programs perform on variables into operations performed on addresses. In general, neither the **name** nor the **type** of the variables survive this translation.

For the most part, programmers don't care about the addresses in which their variables are stored. The compiler picks a location for them that is guaranteed to be satisfactory for as long as the programmer sticks to the rules of the programming language. Some programming languages ensure that the programmer lives by its rules. Java is one such language. But others, of which C is the best example, give the programmer the freedom to do just about anything. But with **freedom** comes **responsibility**. C will let programmers "shoot themselves in the foot" by writing code that subverts the clean abstractions programming languages usually provide.

The primary vehicles by which C enables programmers to do this are the **&** and **\*** unary operators. The **&** operator returns the address in which a variable or expression is stored. The **\*** operator does the reverse: it returns the value stored in the address given to it. As we will see, this can get tricky, because through these operators, a C programmer can make two variable names refer to the same location in memory. Because the hardware cannot distinguish between the two variables in one such pair, an assignment to one of the variables will change the value of the other, and this will not be apparent in the source code.

Let's look at some addresses. [Download](#) the following code and make a new Visual C++ project from it:

```
#include <stdio.h>

char globalchar1;
char globalchar2 = 'g';
int globalint1;
int globalint2 = 9;
char globalchar3;

int main (int argc, char * argv[])
{
    char localchar1;
    char localchar2 = 'l';
    int localint1;
    int localint2 = 1;
    char localchar3;

    printf("Globals: '%c' (%d) '%c' (%d) %d %d '%c' (%d)\n",
           globalchar1, globalchar1,
           globalchar2, globalchar2,
           globalint1, globalint2,
           globalchar3, globalchar3);
}
```

```

        globalchar3, globalchar3);
printf("Locals:  '%c' (%d) '%c' (%d) %d %d '%c' (%d)\n",
        localchar1, localchar1,
        localchar2, localchar2,
        localint1, localint2,
        localchar3, localchar3);

return 0;
}

```

Compile and execute the program. Ignore the warnings regarding uninitialized local variables. You should see output similar, though perhaps not identical, to this:

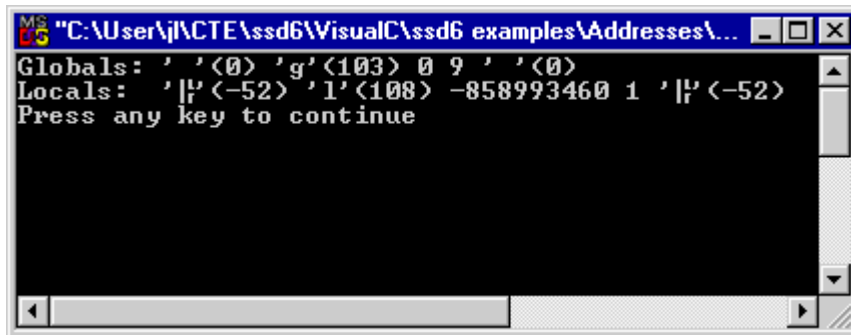

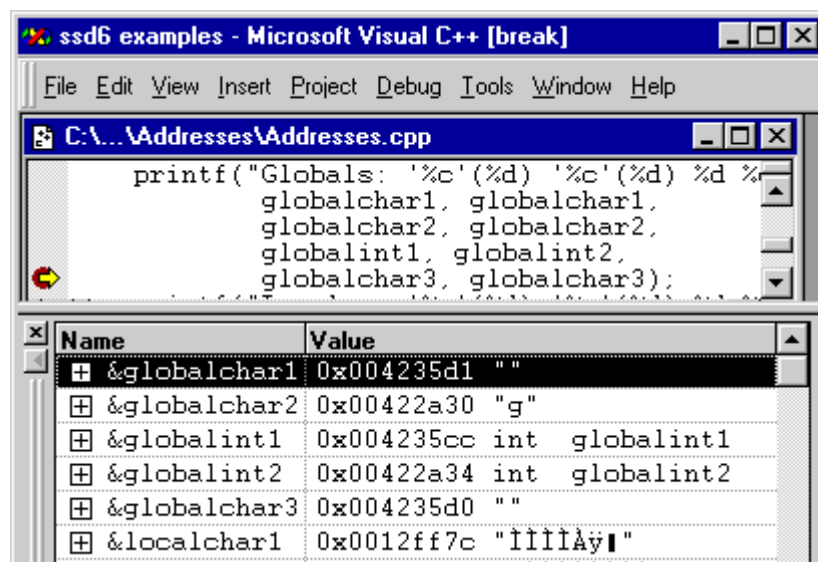


Figure 1: Behavior of the Addresses program

First, notice that some variables are initialized when they are declared, and some aren't. Also notice that some variables are global to the program, and some are local to the main procedure. Finally, note that we have variables of two types: char and int.

Now let's place a breakpoint at the first printf(), and press F5. Let's use the Watch window (accessible through the  icon) to take a look at the addresses of the variables. Remember that the Watch window can track the values of arbitrary C expressions. Since & is a valid C operator, we can ask Watch to track the value of "&variable" in order to track the address of variable. Let's do this now. Get the addresses of all variables displayed in the Watch window now. You will see something like the following (though the actual addresses may not be identical):



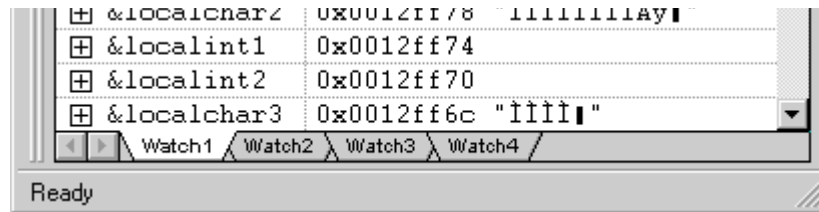


Figure 2: Debugger Displaying Addresses

Addresses are just numbers, but they are often very large numbers. Typical addresses can be as high as 4 billion or, in newer computers, as high as 16 quintillion (which is, roughly, 2 elevated to the 64 power.) Now, the numbers you manipulate normally are expressed in base 10. This means, for example, that the value of the number 20 is equal to  $2 * 10 + 0$ . This is a number expressed in base 10 because we obtain its value by multiplying the 2 by 10. For reasons that will become clear in "[Unit 2. Representation of Data](#)", when dealing with computer-stored numbers, we often prefer to use other number representation bases. Common ones are 2, 8, and 16. A base 2 number is represented with only zeros and ones. A base 8 number is represented with digits 0 through 7. A base 16 number is represented with digits 0 through 9, and letters A through F to symbolize the values 10 through 15. Since we often deal with numbers represented in different bases, we normally use prefixes to indicate the base. A number **without a prefix** is usually assumed to be in **base 10**, a number with a leading **0** is often taken to be in **base 8**, and a number with a leading **"0x"** is taken to be in **base 16**.

In the Watch window you see that the values of all the "&" expressions start with 0x. This means that the value that follows is a number (an address in this case) that happens to be expressed in base 16. There is no mystery beyond that.

Observe that the addresses are **clustered in three groups**, as shown in the figure. Note that **all the local** variables are clustered together. Note also that, of the global variables, those that are **initialized** when declared are in one cluster, and that those that are **not initialized** are in another. Why has the compiler picked these addresses for these variables? Does the pattern reveal anything about how the compiler works? Is this pattern present in all programs?

Look at Figure 1 above, which shows the execution of the program. It would appear that **uninitialized global variables** seem to be somehow **initialized to zero**, but that the same is not true of local variables, which seem to start up with arbitrary values. Can you guess **why** this might be so?

We will partially answer these questions in "[Unit 3. Memory Layout and Allocation](#)" and in "[Unit 6. Operating System Interaction](#)."

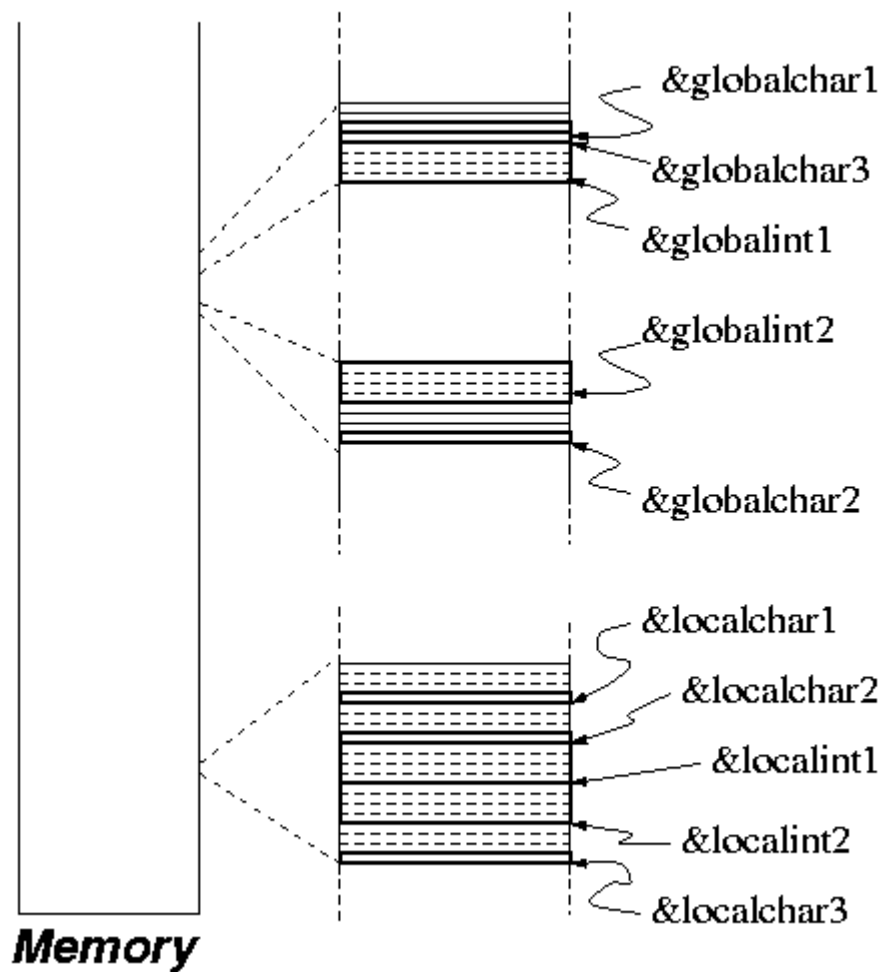


Figure 3: Address Distribution

Observe also that ints take 4 bytes and that chars take only one. Because each byte of memory is independently addressable, one int will “use up” 4 addresses, whereas a char only needs one. For its **convenience**, the compiler may choose to **pack data less densely** than it might. You can see this in the figure, where it is obvious that out of some 4-byte blocks, only one of the bytes is used.

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.

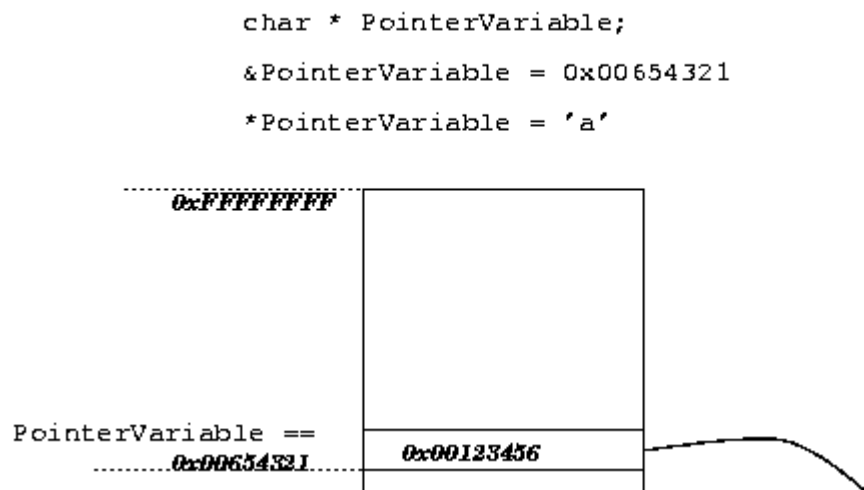
## 1.3.2 Pointers

- Pointers—Stored Addresses
- Memory Allocation
- Data Encoding

### Pointers—Stored Addresses

In the page "[1.3.1 Addresses and Naming](#)" we saw that all data has an address and that an **address** is simply an **integer**. Since memory is able to store integers, we can not only store data in memory but also store the addresses of data in memory. In the latter case, the location where we store an address will itself have an address. C allows us to give the locations of such stored addresses names—that is, C allows us to declare variables that hold the addresses of data, rather than holding data directly. You are probably already familiar with such variable types: they are usually called *pointers* or *references*.

In the figure, `PointerVariable` is a pointer to a `char`. Notice as well that `PointerVariable` is stored at memory address `0x00654321`, and its current value is `0x00123456`. Because memory location `0x00123456` happens to currently contain `'a'`, the value of the C expression `*PointerVariable` is `'a'`.



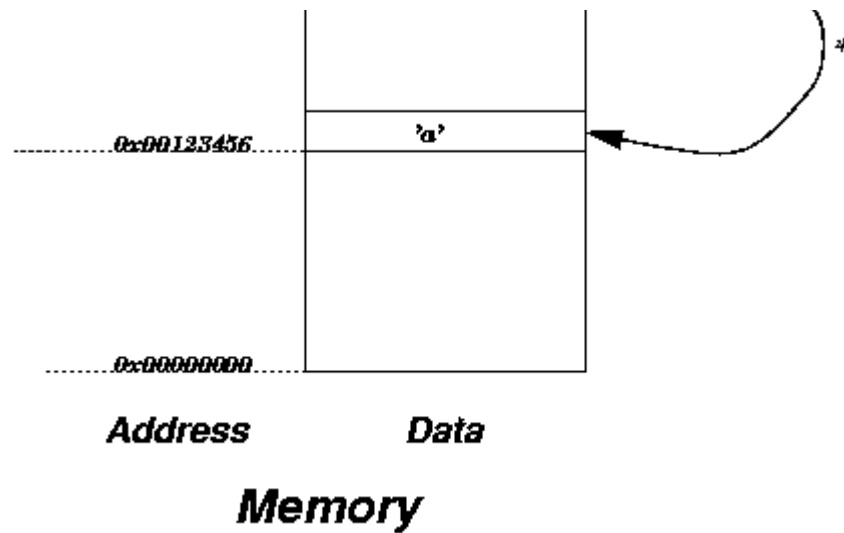


Figure 1: Pointers in Memory

Let's use Visual C++ to look at some pointers. [Download](#) the code below and load it into Visual C++:

```
#include <stdio.h>

main (int argc, char * argv[])
{
    char a = 'a';
    char * ptr = &a;

    while (1);
}
```

Run it under the debugger and stop execution at the line with the while loop. Then look at the values of the following expressions by entering them into the Watch window:

```
&ptr
ptr
&a
a
```

You should have obtained something similar to the



figure below. Take a look at the results, and make sure you understand them.

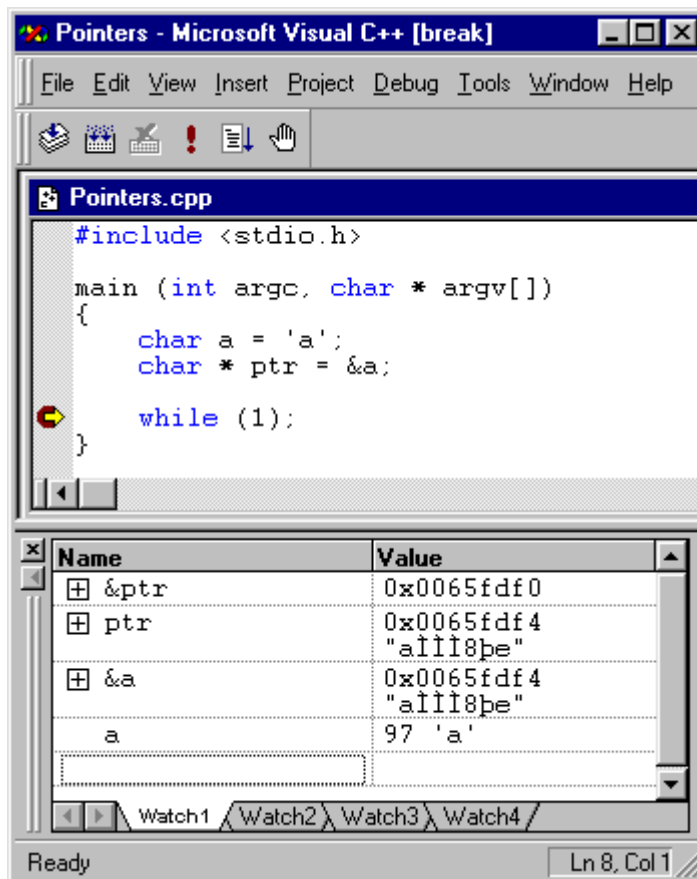


Figure 2: Using Visual C++ to look at pointers

## Memory Allocation

Every item of data resides in some memory location. Even pointers, which are themselves locations, need to reside at some location. In figure 1 above, the value 0x123456 resides in location 0x654321. That address was allocated to it by the compiler because `PointerVariable` was declared as a pointer with the declaration `char * PointerVariable`.

What the pointer points to—in this case, address 0x123456—is usually also allocated by the compiler as a result of some other declaration. In this case, 0x123456 might have been allocated to some variable of type `char` as the result of a declaration such as `char Variable`.

You need to understand that the declaration of a pointer makes the compiler allocate memory to hold the pointer's value (an address), but that the compiler will not automatically also allocate memory for that which the pointer points to. In the figure above, the declaration of `PointerVariable` allocated only the memory at `0x654321`, *not* the memory at `0x123456`. The latter should have been allocated separately by a different declaration.

This has the potential to make trouble. What if the data at `0x123456` is actually allocated as the result of a declaration like `int Variable`? In this case, the expression `*PointerVariable` will be reading an integer as if it were a character!

This is the sort of thing that safe languages like Java do not allow.

## Data Encoding


What does it mean to "read an integer as if it were a character"? Is this possible at all? In fact, it is, because, in the real machine, outside the programming language abstraction, data of any type is represented with nothing but bits. The real machine has no notion of integers or characters, it only knows about bits and bytes. It is the compiler that interprets those bits as integers or as characters. It is also the job of the compiler to write those bits, using an *encoding* that dictates which bit sequence represents which integer, and which bit sequence represents which character. We will cover these encodings in detail in "[Unit 2. Representation of Data](#)," but you should already realize that a character can be read as an integer, and vice versa, because they are both represented with raw bits in the hardware. The results may be surprising. For instance, the encoding of 'A' often happens to be the same as the encoding of the integer 65. This is an arbitrary convention: the number 65 is no more like an

'A' than any other number.

Encoding specifies the mapping of values to bit patterns, but it also specifies *how many* bits a particular type requires. There are far fewer characters than there are integers, so it would be a waste to encode both using the number of bits required for an integer. In fact, an integer occupies more addresses than characters do, to varying extents depending on the computer and the encodings we use. Typical personal computers use 32 bits to represent integers and 7 bits to represent characters in ASCII code. But there are computers that use 64 bits for integers, and the UNICODE character code, for instance, uses 16 bits for each character no matter what computer it is used on. Data types whose representation requires more than one byte will occupy more than one memory address. The compiler takes care of these details for us, at least if we play by the rules.

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.

### 1.3.3 Examining Memory

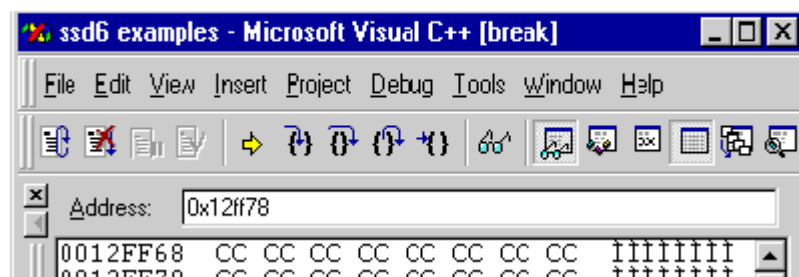
The Visual C++ debugger allows us to look at the raw values in memory. Let's bring up the Memory window. Start up the debugger as before, running on the program `Pointers.cpp`, stopping the program at the while loop. Click on the  icon. The **Memory window** should have popped up. You can achieve the same effect by selecting Memory from the **Debug Windows** submenu in the **View** pull-down menu, and also by pressing **ALT+6**.

The Memory window shows all the values in memory. Values that are **not set** or **not accessible** to the program are shown with **question marks**. Each line shows the contents of several addresses. The first address in each line is printed in the first column. By right-clicking on the Memory window, you get a choice of formats. **"Byte Format"** shows each byte in base 16, and, to the right of the screen, the same bytes decoded as characters. **"Short Hex"** and **"Long Hex"** group the bytes into sets of 2 and 4 bytes, respectively, and show only the base 16 values. Try them.

You can see that memory is very large indeed. To find something of interest, you type the relevant address in the text area at the top of the Memory window. Let's take a look at the address of `ptr`, that is, type the value of `&ptr` into the Address text area. You will need to get the value of `&ptr` from the Watch window.

You should see the address of `a` appear, even though it may be a little hard to recognize, depending on what display format you have selected. The bytes of `&a` may be in **reverse** order. This is all right. How a computer represents multi-byte data types (such as integers) is a matter of convention. While all programs in that computer have to abide by the convention, there is nothing that prevents another computer from encoding integers in any other way. What is important here is that every time your **CPU** encounters that sequence of bytes, it always **interprets** them as the same integer. We will talk more of data representation in ["Unit 2. Representation of Data."](#) Change the display format from **"Long Hex"** to **"Byte"** and then back, and see the difference. When in **"Long Hex"** format, the debugger interprets memory as **integers**, and, therefore, shows you the unperturbed value of `&a`.

Use the same mechanism to look for the value at `&a`. Do you see a 61? If you interpret that byte as a character by again switching the display format to **"Byte"**, it shows as an `a` in the right-hand side of the window.



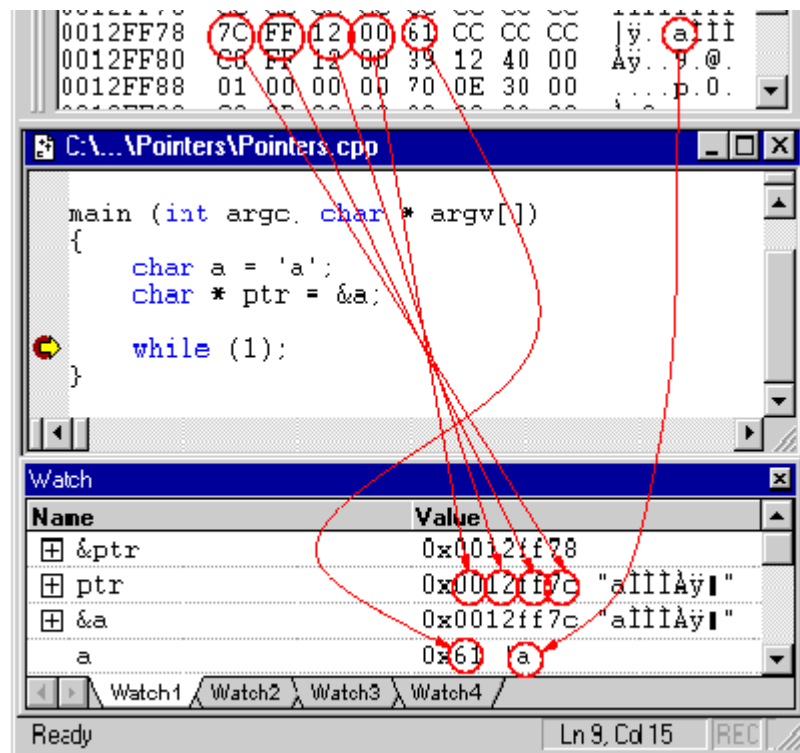


Figure 1: The Debugger Memory Window

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.3.4 Arrays and Strings

- [Arrays Subscripting / Pointer Accessing](#)
- [Allocation and Reference](#)
- [Arrays of Multi-Byte Types](#)
- [Multi-Dimensional Arrays](#)
- [Strings](#)

### Array Subscripting / Pointer Accessing

When we declare an array, the compiler allocates 20 bytes of **contiguous** memory at memory location M for the array declaration `char A[20]` as shown below:

subscript:	[0]	[1]	[2]	[3]	[4]	[5]	· ·	[19]
cell reference:	A [0]	A [1]	A [2]	A [3]	A [4]	A [5]	· · ·	A [19]
memory loc:	M	M+1	M+2	M+3	M+4	M+5	· · ·	M+19

As indicated, the subscripts run from 0 to 19 for array elements (cells) A [0] through A[19]. The address of the array A and its first element, A[0], are identical: memory location M. The store for the array element A[i + 1] immediately follows that for the array element A[i] for  $0 \leq i \leq 18$ .

Arrays may be accessed using subscripting as shown above. Array elements may also be referenced by equivalent pointer references. For example, the subscripted reference A[7] in C is equivalent to the pointer reference A + 7. The pointer is a reference to a character. Hence, the pointer reference has the effect of referencing the character that is 7 characters above the start of array A, and, thus, references 7 characters above the array element A[0]. Instead of writing A[7], we can also write \*(A + 7) directly. This equivalence between subscripted array references and pointer references make the following references equivalent:

A[i]	*(A + i)
&A[i]	A+ i
A[i + j]	*(A + i + j)
&A[i + j]	A+ i + j

### Allocation and Reference

Even though an array can be manipulated using pointer referencing you should not make the mistake of thinking that any pointer can also be thought of as an array.

[Download](#) and execute the following program. This program tries to

initialize two arrays, one to "This is A" and the other to "This is B". It tries to save some work by first initializing the first array, copying it to the second, and then changing only the one character that is different in the two.

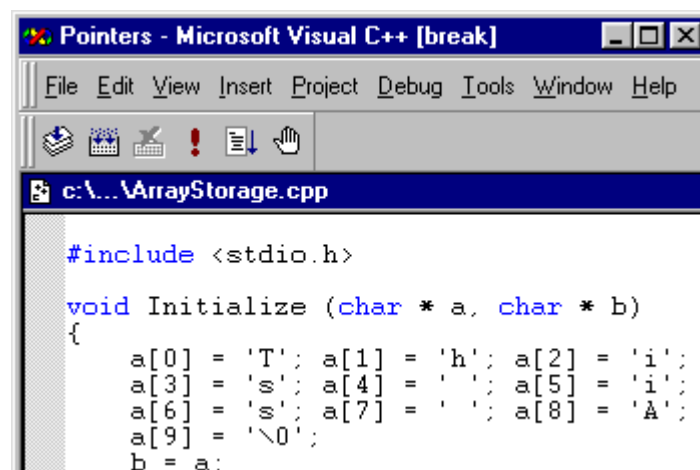
```
#include <stdio.h>

void Initialize (char * a, char * b)
{
    a[0] = 'T'; a[1] = 'h'; a[2] = 'i';
    a[3] = 's'; a[4] = ' '; a[5] = 'i';
    a[6] = 's'; a[7] = ' '; a[8] = 'A';
    a[9] = '\0';
    b = a;
    b[8] = 'B';
}

#define ARRAY_SIZE 10
char a[ARRAY_SIZE];
char b[ARRAY_SIZE];

int main(int argc, char * argv[])
{
    Initialize(a, b);
    printf("%s\n%s\n", a, b);
    return 0;
}
```

When you run this program, **only "This is B" is printed.** To see why, place a breakpoint in the line `b[8] = 'B'` and rerun the program. Take a look at the values of `a` and `b`. They are the same, as they should be, because we assigned `a` to `b` in the previous line. What is the same, however, is not the *contents* of the arrays but the two arrays have the **same address** after the assignment `b = a`. The contents are not only equal, they *are* the same. Modifying one will also modify the other. The assignment `b[8] = 'B'` will modify the bits of memory that also contain `a[8]`, and the contents of the original `b` will remain untouched. The two original arrays still get printed, but the **first** one will now contain **"This is B"** and the **second** will be **empty** because it was never touched. The address of the second one, which was allocated by the compiler when it was declared, was overwritten and discarded by the line `b = a` before its contents were modified.



```
#include <stdio.h>

void Initialize (char * a, char * b)
{
    a[0] = 'T'; a[1] = 'h'; a[2] = 'i';
    a[3] = 's'; a[4] = ' '; a[5] = 'i';
    a[6] = 's'; a[7] = ' '; a[8] = 'A';
    a[9] = '\0';
    b = a;
}
```

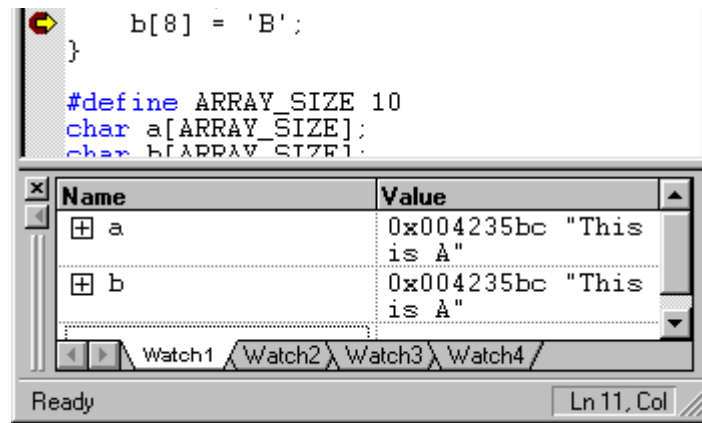


Figure 1: Running ArrayStorage.cpp

This example illustrates what we meant when we say that **C has no true arrays**. A language with true array support would interpret the line `b = a` as **copying the contents** of `a` into `b`, much like it would if `a` and `b` were integers (as opposed to the assignment of `a`'s address to `b`). A banal assignment like that would not result in two variables invisibly interfering with each other. Instead, we would expect the behavior shown below for true array support.

True array support would have the compiler generate object code to copy the contents of array `a` to array `b`. The generated object code would perform the operations shown below in this C code. It would need to create a loop that copied, one by one, the contents of each character from array `a` to array `b`:

```
void Initialize (char * a, char * b)
{
    int i;
    a[0] = 'T'; a[1] = 'h'; a[2] = 'i';
    a[3] = 's'; a[4] = ' '; a[5] = 'i';
    a[6] = 's'; a[7] = ' '; a[8] = 'A';
    a[9] = '\0';
    for (i = 0; i < ARRAY_SIZE; i++)
    {
        b[i] = a[i];
    }
    b[8] = 'B';
}
```

The loop would be rather more expensive than the apparently inoffensive `b = a` might lead us to believe. If you cared about **performance**, you might prefer to use a language that made time-consuming operations require more code. That is, you might prefer a language that let you program at a **lower level of abstraction**, a language that expressed more closely what the hardware can do.

## Arrays of Multi-Byte Types

So far, we have only dealt with arrays of characters. Since a character typically occupies one byte, and each byte occupies one address, the



last byte of the second integer because integers occupy four bytes each.

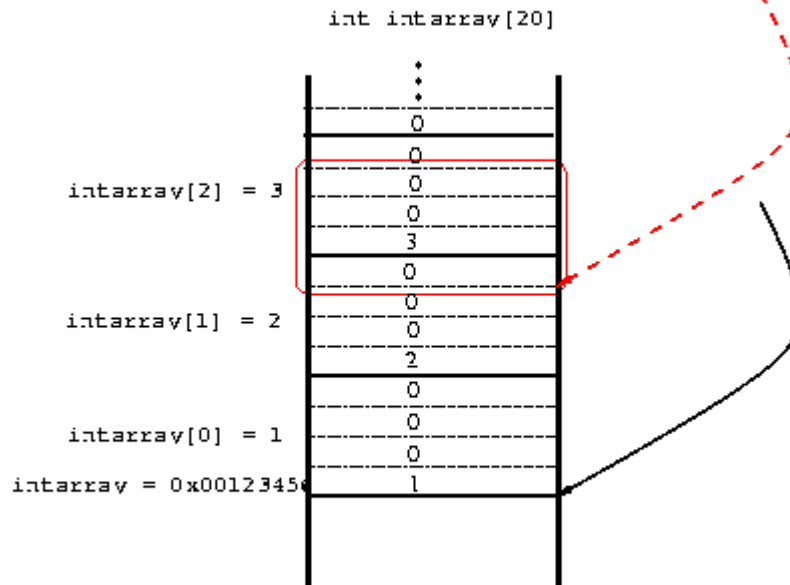
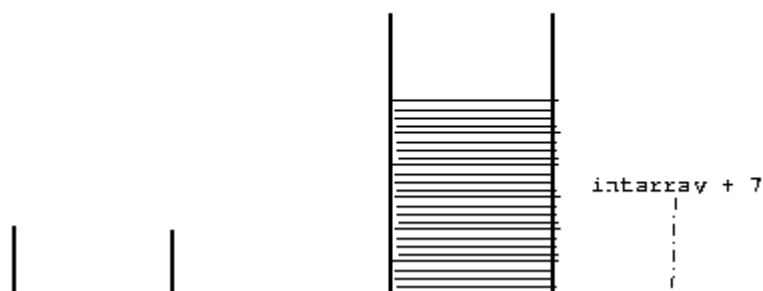


Figure 2: Arrays of multi-byte types

But C finds it so useful to allow the programmer to manipulate arrays as if they were pointers that it treats **pointer addition** differently from integer addition, in order to maintain the above expression equivalency for arrays of multi-byte types. In C, the expression `intarray + 7` yields an address that is  $7 * 4$  bytes, rather than 4, above the address of `intarray`. The **compiler** does this because it knows that `intarray` is a pointer to an integer, that integers are four bytes, and that each byte consumes one address. In doing so it appears to redefine the rules of arithmetic. In fact, arithmetic on pointers does not follow strictly the rules of normal arithmetic, when we take the pointers to be equivalent to the addresses they represent. As we saw, **adding 7 to a pointer** might yield a value that is **28 bytes larger!** For this reason it is usually called **"pointer arithmetic."**



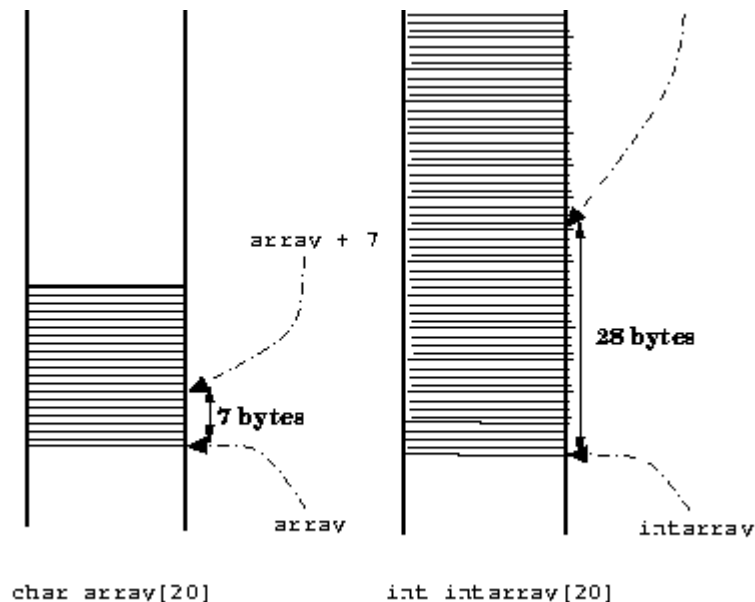


Figure 3: Integer and Character Pointers

This oddity of computing with pointers is useful because it allows us to manipulate arrays using pointer expressions, no matter what the type of the array contents. **Pointer expressions** are easier to read, once you get used to them. They are also less abstract, that is to say, they are closer to how the hardware deals with arrays. This also means that they can be faster than the equivalent array expressions.

We have seen what it means to **add an integer to a pointer: the integer is scaled (multiplied) by the size of the pointer-target type.** The same is true for **subtraction.** That is, the expression `intarray - 7` will yield an address that is 28 bytes below that of `intarray`. This is very lucky, because it means that `intarray + 7 - 7` is equal to `intarray`—which we would expect and which would not be true if the scaling was applied only to the addition.

In all this, we have added and subtracted an integer to a pointer. Do you think it might be possible to **add and subtract two pointers?** Think about what that would mean and whether it would be useful, and then compare your conclusions to what the C programming language stipulates regarding such operations.

## Multidimensional Arrays

Consider the 2-dimensional array declaration below:

```
int md[3][4]
```

The array name MD (for multi-dimensional) denotes an array of three elements. These are allocated contiguously. Each such element is an array of four int's. Again, each of the four int's area allocated contiguously. The Visual C++ compiler guarantees the **row-major ordering** of all elements in memory. The following table illustrates the relationship between the storage for array elements and the subscripts that access each array

element where their storage for array MD begins at the memory location M:

array name:	MD	MD	MD	MD	MD	MD	MD	MD	MD
row index:	[0]	. . .	[0]	[1]	. .	[1]	[2]	. .	[2]
col index:	[0]	. . .	[3]	[0]	. .	[3]	[0]	. .	[3]
mem loc:	M	. . .	M+12	M+16	. .	M+28	M+32	. .	M+44

One way to access all elements of the two-dimensional array is to use subscripting as follows:

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; j++)
        md[i][j] = 0;
```

Another, equivalent method to access these array elements is to use pointer referencing as follows:

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; j++)
        *((int*)(md) + i*4 + j) = 0;
```

Thus, the concept of equivalencing subscripting and pointer arithmetic holds for multidimensional array as it does for one-dimensional arrays.

## Strings

The difference between a string and a character array in a language like C++ is, mainly, that **strings are of variable length.** In C, there is no string type. C programmers use **character arrays** when they need a string and receive much help in doing this from the library functions that are available in most environments. Examples of these functions are printf(), strcat(), and strcmp(), which provide the illusion that the character arrays they manipulate are actually of variable length.

To treat a fixed-size character array as a variable-length string, these library functions need to keep track of where **the last character** of the string is within the array. They could do this by attaching an **index** to the array that would contain the position of the last character of the string that the array represents.

Rather than storing such an index alongside a character array, these functions manage to implement variable length strings without any extra storage. They do this by adhering to the convention that a string ends at the first location in the array that contains a **"null character,"** that is, the character whose memory encoding is zero. This means that **strings**

cannot contain the null character, but this is all right, since the null character does not have a valid representation in text.

Because strings are implemented as arrays, and arrays are implemented as pointers, we can manipulate strings by manipulating pointers using pointer arithmetic. The following example takes some input and breaks it down into words, using character pointers and the `strchr()` library function to find the spaces that separate the words. You might want to [download](#) it and examine the values of the relevant variables as it executes.

```
#Include <stdio.h>
#include <string.h>

#define MAXLINE_LEN 80

int main ( int argc, char * argv[] )
{
    char Buffer[MAXLINE_LEN];
    char * space;
    char * word;

    printf("Input> ");
    do {
        if (fgets(Buffer, MAXLINE_LEN, stdin) == NULL) break;
        word = Buffer;
        while (space = strchr(word, ' '))
        {
            *space = '\0';
            printf("%s\n", word);
            word = ++space;
            while (*word == ' ') word++;
        }
        if (strchr(word, '\n')) {
            printf("%sInput> ", word);
        }
        else {
            printf("%s\n", word);
        }
    }
    while (strcmp(Buffer, "exit", 4));
    return 0;
}
```

We will talk again about arrays and strings in C when we examine how the various data structures are laid out in memory. We will do this in "[2.4.1 Strings and Arrays](#)."

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

### 1.3.5 Naughty Pointers

We have seen that a pointer is nothing more than an address stored in memory and that the address carries no information about *what* is stored at the location to which it refers. When memory is accessed by dereferencing a pointer (for example, by the expression `*PointerVariable`), its contents are interpreted by the **compiler** according to the declaration of the pointer, not according to what the memory actually holds. For example, we saw that when `PointerVariable` was declared as a `char *`, the data at the address was interpreted as a character even if the variable that resided there had been declared an integer.

It is also true that if a pointer seems to refer to part of an array, the compiler will interpret the data to which it points as if the data were part of the array, whether or not the data actually *is* part of the array.

For example, in the page "[1.1.1 Execution as Physical Process](#)," we showed the code below, the execution of which **never ends**:

```
#define ARRAY_SIZE 10

void natural_numbers (void) {
    int i;
    int array[ARRAY_SIZE];

    i = 1;

    while (i <= ARRAY_SIZE) {
        array[i] = i - 1;
        i = i + 1;
    }
}
```

As we briefly explained, the problem is that **array[10]** resolves to the same memory address as does **i**, because the **compiler** happens to **allocate** **i** immediately above the 10 integers allocated to **array**. Then **array[10]** is translated by the compiler into **array + 10**, which is in fact the address of **i**. Therefore, when the program assigns **i - 1** to **array[10]**, the hardware is actually decrementing **i**, ad infinitum. This could happen because, when the compiler treats **array[10]** as a pointer, it does not check that the resulting address is, in fact, within the extent of the memory block allocated to the array. The compiler inadvertently assigns a value to the location that holds **i** because it blindly trusts the programmer to use pointers judiciously.

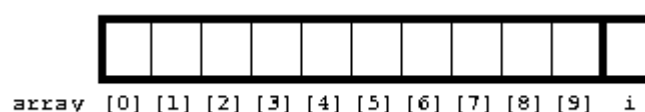


Figure 1: Writing **array[10]** **overwrites** **i**.

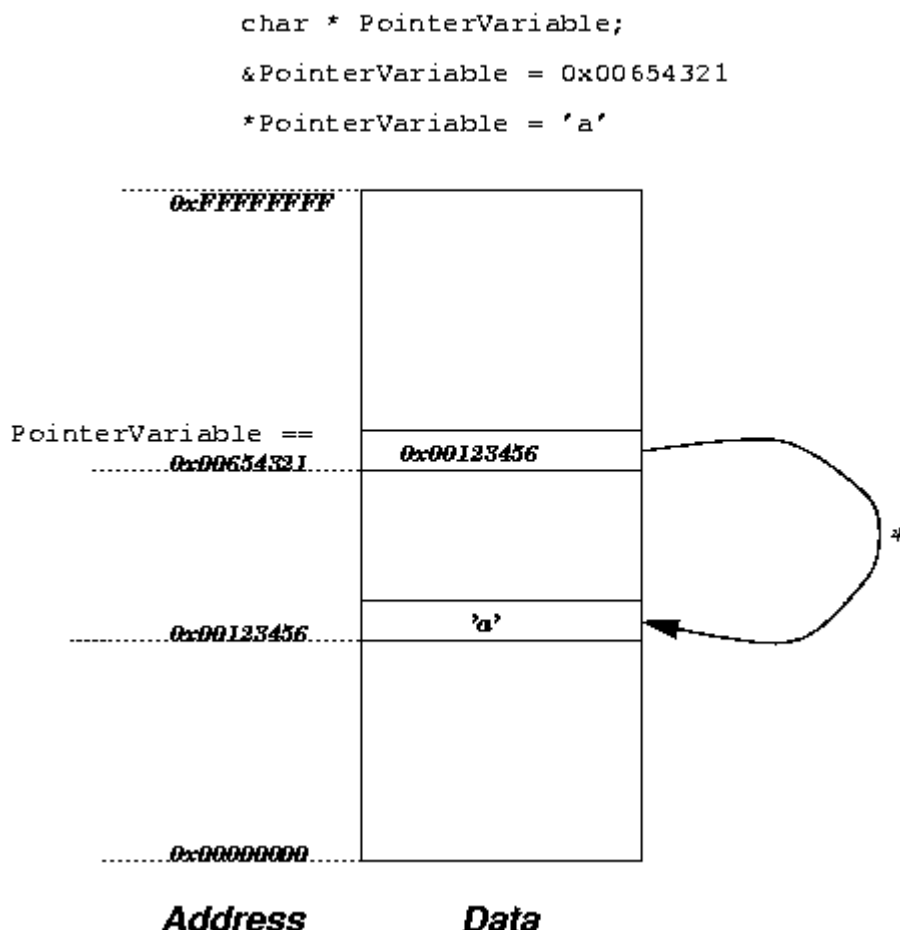
Such use of pointers is not always an accident. Some times we actually

want the **freedom** to access any part of memory and to **interpret** it as we wish, regardless of what the compiler thinks that memory actually holds. This is especially so when we are programming systems at a low level and need to, for example, interact with hardware devices.

Through a combination of the \* operator and type "casting," C allows us to look at the contents of any address in memory and to interpret the raw bits therein as we wish.

A type cast is a compiler directive that **forces the compiler to interpret** data of one type as if it were data of another type. For example, the value of the expression (char) 65 might be 'A', because 65 is the integer that encodes the character 'A' in the ASCII character set. **The expression (char) is the cast,** which forces the compiler to interpret 65 as a character type, even though it is obviously an integer. We can cast any pointer type to any other pointer type, simply by replacing the "char" within the parentheses with whatever type we wish to cast into.

Because an address is nothing more than an integer, and we can dereference addresses with the \* operator, we can look at the contents of memory address X by casting the integer X to be a pointer, and then applying the \* operator to the result. In the example of Figure 2, the value of the expression \* ((char \*) 0x00123456) would be 'a', because address 0x00123456 contains the value that corresponds to 'a' in the character code we are using.



## Memory

Figure 2: Previous example of a simple pointer in memory

Let's now use type casts to look at the contents of specific memory addresses. Open up the following sample program in Visual C++. You probably already have it somewhere, but you can [download](#) it again if you don't.

```
#include <stdio.h>

main (int argc, char * argv[])
{
    char a = 'a';
    char * ptr = &a;

    while (1);
}
```

As before, put a breakpoint in the line with the while, press F5, and look at the values of a, &a, ptr and &ptr.

Now, the bad stuff. Suppose that in the "Watch" pane you see that the value of &a is 0xAAAAAAAA, and that the value of &ptr is 0xPPPPPPPP. Set watches for the following two expressions, in which you would replace the AAAAAAAAA and PPPPPPPP by whatever values you observed:

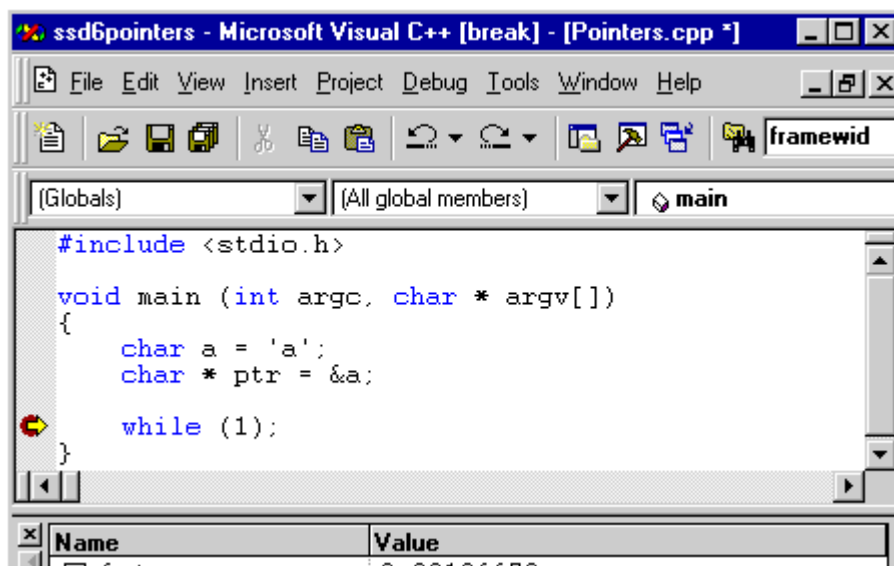
```
* ((char *) 0xAAAAAAAA)
* ((int *) 0xPPPPPPPP)
```

You should see that the two later expressions access the same values as the earlier ones, even though they contain no variable names.

Now, set a watch for the following expression:

```
* ((char *) 0xPPPPPPPP)
```

The result should be similar to this:



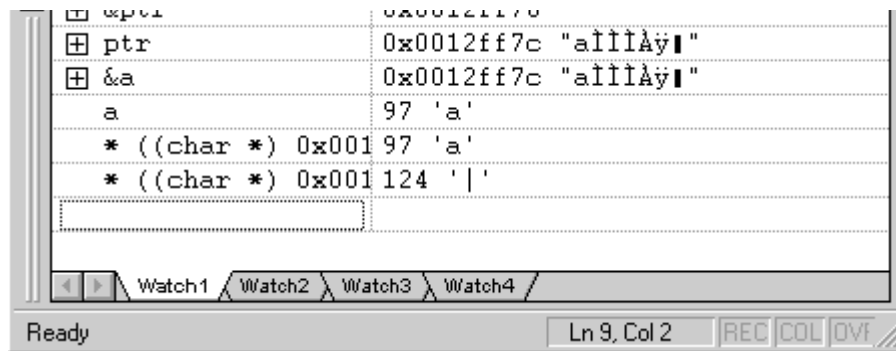


Figure 3: Values of Pointers

You know that the value at &ptr is &a: it is assigned to that in the second line of the program! Why are we getting a small integer instead? It may help to clarify matters to display this value in base 16. To do so, right-click in the Watch window and set the **Hexadecimal** option. This results in something like this:

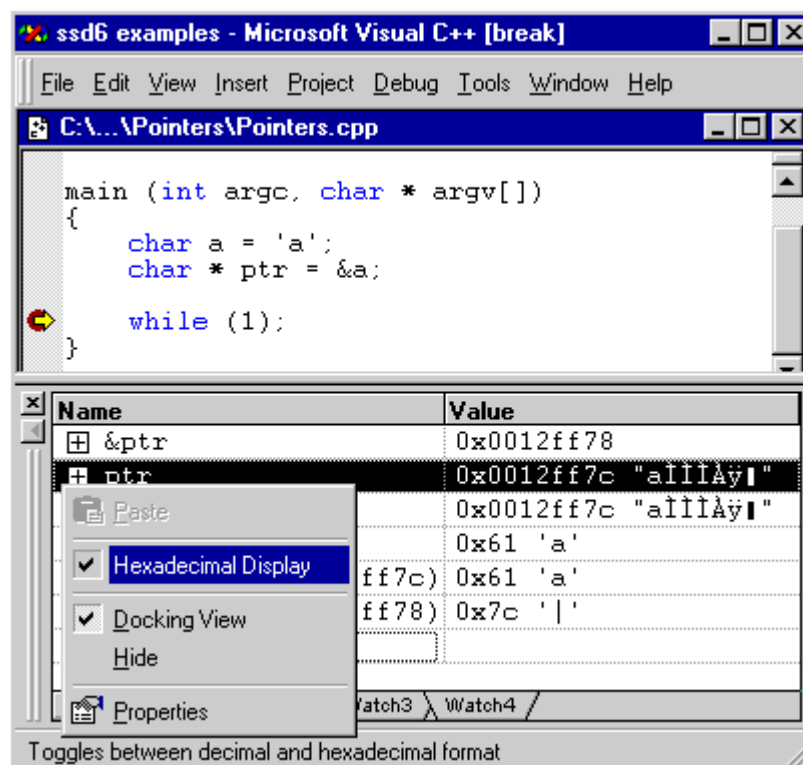


Figure 4: Setting Values to Display in Base 16

Now you see that the value at the address &ptr is actually the **same** as the least significant byte of &a, which is at least close. The problem here is that the expression `* ((char *) 0xPPPPPPPP)` interprets the data at address 0xPPPPPPPP as a **character**, even though it is a pointer. Because characters occupy only **one byte**, the value of the watch expression is **only the first byte** of the value of ptr, even though ptr is a four-byte address.

Do you understand how casting and pointers interact in C? If you do, you *really* understand memory addresses.

The lesson here is that, as far as the hardware is concerned, memory is a



featureless sequence of bytes that gets interpreted as single bytes or as groups of bytes, depending on how the compiler is trying to interpret them. The same underlying bytes can be interpreted as one int or four chars. The C compiler usually keeps all this straight for us, interpreting memory locations the right way. But if we circumvent the safeguards of the compiler, as we did by using a cast, the compiler can no longer guarantee the correct interpretation of memory.

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.

## 1.4 Data and Function Calls

- 1.4.1 [Implementation of Local Variables](#)
- 1.4.2 [Activation Records and Stacks](#)

### Assessments

- [Multiple-Choice Quiz \(4\)](#)

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.4.1 Implementation of Local Variables

- [Scopes](#)
- [Formal Parameters: Value and Reference](#)
- [Recursion](#)

### Scopes

Previous programming experience has already made you familiar with the notion of *scoping*. **Variables have scopes**, that is to say, a variable name may refer to different values depending on where it is used. If used inside a function that declares a local variable of that name, the name will refer to a data item that is private to that function and not a variable of the same name declared in another function or globally.

If a name can refer to different values, it must also be able to refer to two different memory locations. Compilers implement scoping by manipulating different memory addresses from within different scopes. The same variable name may be mapped to different addresses by the compiler. The hardware does not know about scoping, just as it doesn't know about variable names or types. The hardware merely manipulates memory addresses, blindly. Let's take a look at the addresses of some local variables. [Download](#) the following program and load it into Visual C++:

```
#include <stdio.h>

int first;
int second;

void callee ( int first )
{
    int second;

    second = 1;
    first = 2;
```

```

    first = 2;
    printf("callee: first = %d second = %d\n", first, second);
}

int main (int argc, char *argv[])
{
    first = 1;
    second = 2;
    callee(first);
    printf("caller: first = %d second = %d\n", first, second);
    return 0;
}

```

Put in watches for &first and &second. Place a breakpoint in the line `callee(first)` and press F5. Note the address of the variable `second` for later reference (we will call it 0xSSSSSSSS—in Figure 1, 0xSSSSSSSS would be 0x004235c0). At this point, the name `second` refers to the global variable of the same name.

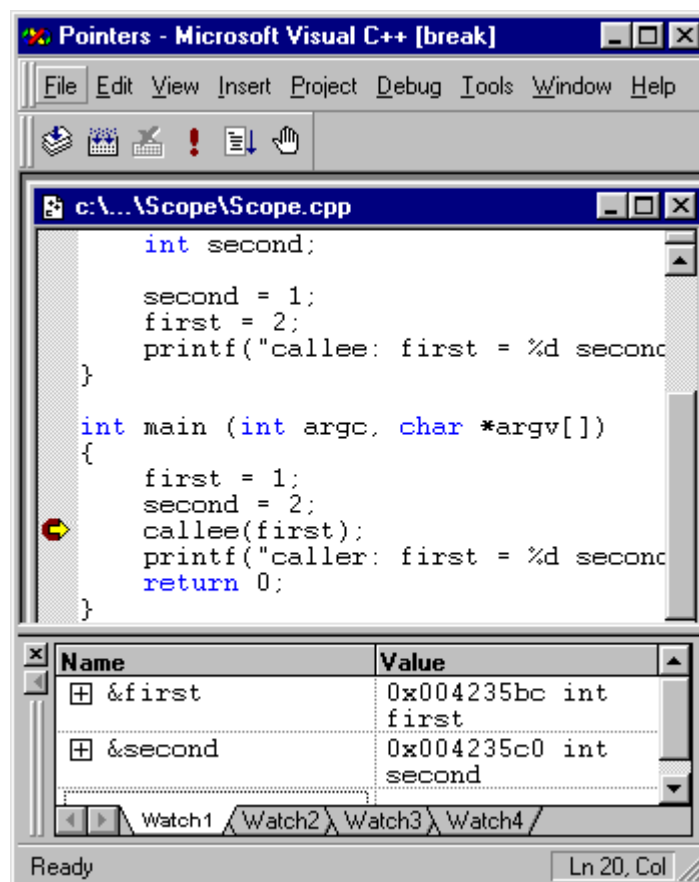


Figure 1: Locations of global variables

Now place another breakpoint in the line `first = 2` and

press F5 again. Now the address of second has changed dramatically. This is because, now, we are looking at the address of the local variable of that name, which as far as the hardware is concerned, is an entirely different variable. The compiler (and the debugger) maps the same name to completely different addresses in order to achieve the illusion of scoping.

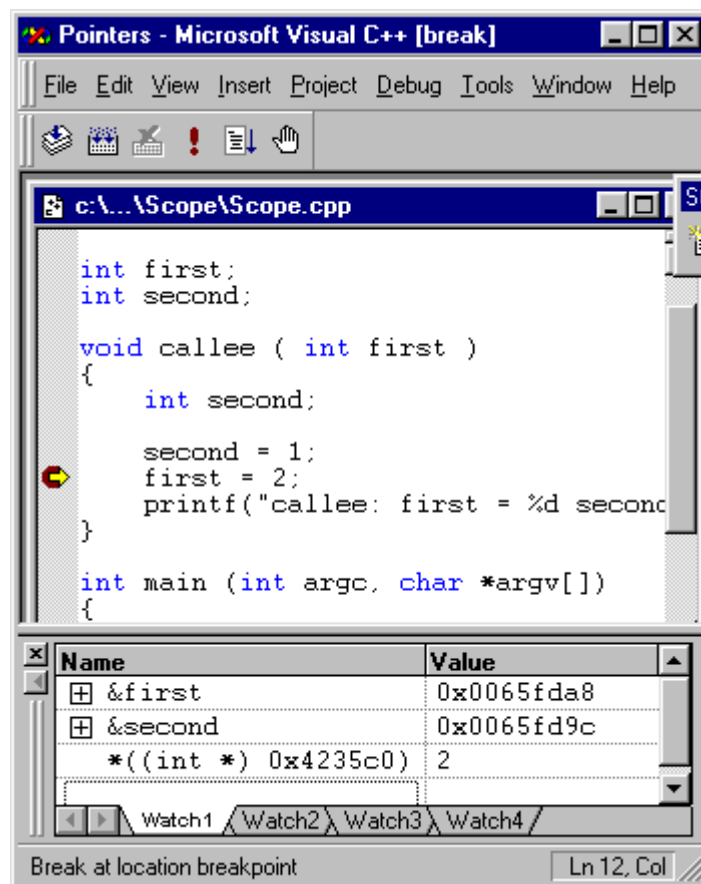


Figure 2: Locations of local variables

Even though we have changed the value of the local instance of second by assigning it to 1, the global instance of second is able to keep its value because it resides in an entirely different memory location in which the assignment to the local second had no effect. Let's verify that this is so by putting in a watch for \*((int \*) 0xSSSSSSSS). You should see that the value at the address where the compiler has allocated the global second is still 2, even though the value of the local second has been set to 1.

## Formal Parameters: Value and Reference

While looking at the value of `&second` above, you may have noticed that the value of `&first` also changed. This might not be what you expect: after all, `first` is not a local variable but a formal parameter to a function. But, as you confirmed, the compiler assigns it an address for the scope of the callee—just as if it were declared locally. In fact, you can see that its address is very near that of the local `second`.

That happens because `first` is a parameter passed *by value*. This means that the value of the parameter is passed down, but that any modifications made to the parameter are *local* to the callee. The scope of a parameter that is passed by value is identical to the scope of a variable that is local to the callee. The compiler implements them both by assigning an address that is different from the addresses of global variables and actual parameters. This allows the actual parameter to remain unperturbed by any assignment made to the formal parameter in the callee.

Parameters to functions can also be *passed by reference*. In this case, the compiler would not allocate a new address for the formal parameter, but would use the address of the parameter specified by the caller. This way, an assignment to the formal parameter within the callee would end up changing the value of the variable passed by the caller.

C has no language support for calling by reference, but it is very easy to achieve the same effect using its `&` and `*` operators. If, in the program above, we wanted to achieve the effect of passing the parameter by reference, we would write something like this:

```
#include <stdio.h>

int first;
int second;

void callee ( int * first )
{
    int second;

    second = 1;
    *first = 2;
    printf("callee: first = %d second = %d\n", *first, second);
}

int main (int argc, char *argv[])
{
    first = 1;
    second = 2;
    callee(&first);
    printf("caller: first = %d second = %d\n", first, second);
    return 0;
}
```

Notice that, in this version, instead of passing the argument, we pass its **address**, and that, within callee, instead of manipulating the formal parameter, we manipulate the integer that the formal parameter points to (by manipulating `*first` rather than plain `first`).

## Recursion

We haven't yet talked about **how** the compiler decides **where** to allocate variables. In the example above we saw that local and global variables seem to be allocated in very different address ranges, and also that globals are near other globals, and locals near other locals.

We are not so interested in knowing the precise details of how the compiler decides where to put each variable—each compiler does it differently, and the specific method is complicated and not very enlightening. But we should be curious about why local variables and globals are allocated in such different places. The answer will come

as we examine recursive function calls, such as the following:

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

void callee (int n)
{
    if (n == 0) return;
    printf("%d (0x%08x)\n", n, &n);
    callee (n - 1);
    printf("%d (0x%08x)\n", n, &n);
}

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

    if (argc < 2)
    {
        printf("USAGE: %s \n", argv[0]);
        return 1;
    }

    n = atoi(argv[1]);

    callee(n);
    return 0;
}
```

[Download](#) the code and load it into Visual C++. This program is the first program we write that will require a command-line argument. To run it from within Visual C++ with an argument, go to the menu **Project:Settings** and click on the **Debug** tab. Set **Program arguments** to some number (10, for example), and run the program by pressing CTRL+F5. You will see something similar to the following:



```
10 (0x0065fda4)
9 (0x0065fd4c)
8 (0x0065fcf4)
7 (0x0065fc9c)
6 (0x0065fc44)
5 (0x0065fbec)
4 (0x0065fb94)
3 (0x0065fb3c)
2 (0x0065fae4)
1 (0x0065fa8c)
1 (0x0065fa8c)
2 (0x0065fae4)
3 (0x0065fb3c)
4 (0x0065fb94)
5 (0x0065fbec)
6 (0x0065fc44)
7 (0x0065fc9c)
8 (0x0065fcf4)
9 (0x0065fd4c)
10 (0x0065fda4)
```

This shows that the compiler allocates one address for each call to callee—in this case, 10. This is the right thing, because `n` is passed by value.

But how did the compiler know that it had to allocate 10 instances of `n`? We could have given the program a totally different argument: 1, 30, or 20,000,000. In each case the compiler would have to allocate 1, 30, or 20,000,000 instances of `n` respectively. And the compiler would not be able to decide how many to allocate until after the program was already running. But if the program is running, the compiler has already finished its work!

What happens is that the compiler inserts additional code for every function call and every function return. This code allocates any local variables that the callee needs for that invocation. Multiple invocations of the callee activate this allocation code over and over again. This is called *dynamic allocation*, because the local variables are allocated at *runtime*, as needed.

Global variables can be allocated *statically*. This means that the compiler can **fix** specific addresses for global variables **before the program executes**. But because functions can be called recursively, and each recursive invocation needs its own instantiation of local variables, compilers must allocate local variables dynamically. Such dynamic behavior makes the program run more **slowly**, so it is not desirable. But it is necessary for local variables.

Because the compiler does not know how many variables it will need to allocate dynamically, it **reserves a lot of space for expansion**. This is why the local variables are allocated to addresses that are **very far away from** the addresses that hold the global variables.

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.

## 1.4.2 Activation Records and Stacks

In the previous page "[1.4.1 Implementation of Local Variables](#)," we saw that local variables have to be allocated dynamically. This slows down the execution of the program a little bit, but it is necessary if the programming language is to allow recursive function calls.

To minimize the cost of dynamic allocation, the compiler tries to allocate local variables in large groups, to amortize the cost of a single allocation over many variables. For each function, the compiler calculates the amount of total space required for the function's locals and allocates that space in one single chunk. The calculation is done at compile time, when it does not slow down the program's execution, and it results in a single allocation operation at run time.

To see how this might be better, suppose you are a travel agent that books flights for groups of vacationers. You might ask each customer to come at a different time, or you might ask them to come all together. If they come separately, for each of them you will need to log onto the reservation system, look for one available seat, and make the reservation. If they come together, you can request a single chunk of seats from the airline with only a little more work than you would need to do for each single person. This also means that they will be more likely to sit together.

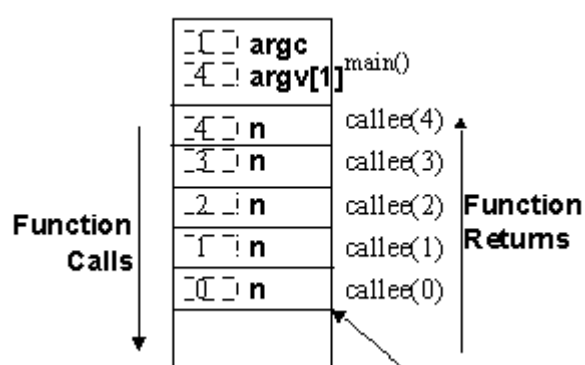
The chunk of memory allocated for each function invocation is called an *activation record*. An activation record for a function invocation is created when the function is called, and it is destroyed when the function returns. Because this operation is so common, computer hardware often provides special support for it. Since hardware can only do simple

things, the activation records need to be organized in a simple manner.

Activation records are organized in a *stack* (as in, for example, a stack of plates in a kitchen). For this reason, they are often also called *stack frames*. The activation record for *main* is at the bottom of the stack, and, as one function calls another, the activation record for *the callee* is *pushed on top of that of the caller*. Only the activation record at the *top of the stack* (or simply TOS) can be *accessed*: to access one below, we need to first *pop* the activation records that are on top of it. This restriction may seem severe, but it keeps things *simple*, and turns out to be sufficient. That it makes things simple is good because it makes it possible for the hardware to help. It is *sufficient* for two reasons:

- A function never returns before *all* of its active *subroutines* *return*—that is to say, we never need to delete an activation record that is not at the top of the stack.
- The rules of scoping imply that a function *cannot* *access* the local variables of its *parent*—that is to say, we never need to access data from an activation record that is not at the top of the stack.

✓ In practice, most computers allocate activation record stacks (often called simply “stacks”) *from higher* addresses *to lower* addresses, and not from lower to higher as the metaphor of “stacks” would lead you to expect (making stacks of plates, for example).



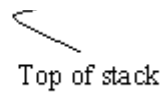


Figure 1: A stack for the previous example

At any point during execution, the compiler and hardware internally maintain two important values that serve to delimit and manipulate the activation records in a **simple** yet **elegant** manner:

- The **stack pointer** holds the address where the stack ends—it is here that a new activation record will be allocated.
- The **frame pointer** holds the address where the **previous** activation record ends—it is to this value that the stack pointer will return when the current function returns.

When a function is called, the compiler and hardware:

- Push the frame pointer into the stack.
- Set the frame pointer equal to the stack pointer.
- Decrement the stack pointer by as many memory addresses as are required to store the local state of the callee.

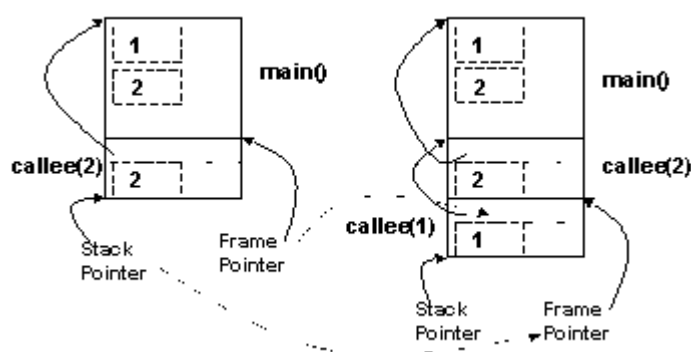


Figure 2: Stack "push" (on function call)

When a function returns, the compiler and hardware:

- Set the stack pointer equal to the frame pointer.
- Pop the value of the old frame pointer from the stack.

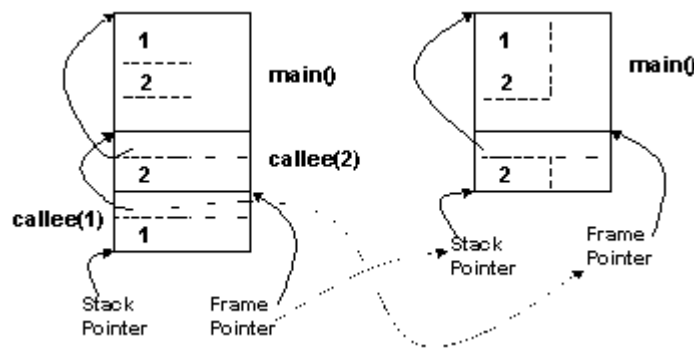


Figure 3: Stack "pop" (on function return)

By so manipulating the stack pointer and the frame pointer, the potentially expensive task of dynamically allocating local variables becomes **simple, well organized,** and **fast.**

You should notice that in describing what goes on when a function is invoked and when it returns, we added an item to the activation record, namely, the old frame pointer that allows the old activation record to become active upon return. Activation records, in fact, contain several things other than **local variables.** They contain **return addresses** and **transient temporary values of the caller** as well. We will learn more about these in the next module "[1.5 The Code Itself.](#)" Activation records also contain the **arguments** to the callee. We showed this implicitly by showing that the argument to the recursive function above, `n`, was in fact in the activation record.

Arguments to functions are **deposited** by the **caller** and **retrieved** by the **callee** from a **stack location** that is fixed with respect to the stack pointer or the frame pointer. All compilers for a particular type of hardware **follow this convention.** This makes it possible for a function compiled by one compiler to call a function compiled by another one. If you were to use a compiler other than Visual C++, for instance, your programs could still make calls to pre-compiled library functions such as `printf()` because both the caller and

the callee would be able to transfer the arguments through their standard locations in the activation record.

We will talk about stacks in more detail in ["3.1.2 Dynamic Allocation."](#)

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.

## 1.5 The Code Itself

- 1.5.1 [Code Is in Memory, Too!](#)
- 1.5.2 [Code Has Addresses, Too!](#)
- 1.5.3 [Registers](#)

### Assessments

- [Multiple-Choice Quiz \(5\)](#)

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.



## 1.5.1 Code is in Memory, Too!

- [The Code in Memory](#)
- [Assembly Code](#)
- [Assembly vs. C](#)

### The Code in Memory

In the preceding pages we went through some pain to detail what **data** looks like in memory and where it is placed, but we didn't say anything about **code**. Compilers translate program statements into detailed instructions to the CPU. We should also ask what these instructions look like and where they reside.

The instructions that the CPU understands are of very limited power. A CPU understands instructions such as:

- "Move the byte at address X into location Y."
- "Add the contents of these two addresses together and put the result in Z."
- "Is the byte at address X **zero**?"

A CPU doesn't understand instructions such as:

- "Calculate the address of the fifth element of this array."
- "c = (a + b) \* d"
- "while (i < MAX)"

The latter are too complicated for the CPU. The CPU design concentrates on doing a few **simple** things really **fast**, and leaves the **complexity** and **flexibility** to the compiler.

The CPU executes one such simple instruction at a time. That instruction is remembered, temporarily, in the circuitry of the CPU. The rest of the instructions, those that are not currently being executed, reside in memory. In modern computers, they reside in the same memory in which data resides. In fact, in memory, instructions are **indistinguishable** from data, because, **in memory, everything is represented as raw bits.**

We saw that both integers and characters were encoded into raw bits. We saw that the encodings for these two data types were different, and resulted in bit patterns of different sizes. Instructions are also encoded into bits. And just like an integer could be accidentally read as if it were a character and vice-versa, so can instructions be accidentally read. We can look at instructions the same way we used pointers to look directly at specific memory addresses.

Let's take a look at an example. Look at the following simple program:

```

#include <stdio.h>
int a;
int main (int argc, char *argv[])
{
    int i; unsigned char * c;
    a = 5;
    c = ((unsigned char *) 0x0040b7D8);
    for (i = 0; i < 10; i++)
    {
        printf("%02X ", *c);
        c++;
    }
    printf("\n");
}

```

This program **prints a part of itself**. Using the techniques you already know (casts and pointer dereferences), it prints the value of the raw bytes of memory starting at address 0x40B7D8. This address turns out to be the address where our version of Visual C++ placed the instruction that implements the assignment **a = 5**. We will soon show you how we found that out.

Using the Visual C++ debugger we also found out that the compiler allocated `a` to address 0x004225D8. Please note that the exact values may be different when you compile the program.

This is the output of the program above:

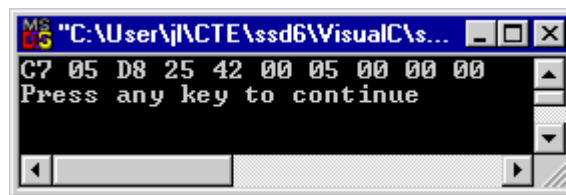
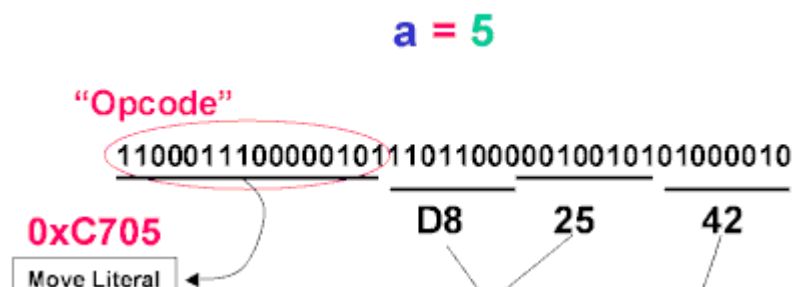


Figure 1: Output of the program above

You can see that even though the address 0x40b7d8 contains a CPU instruction, we can read it as if it were a bunch of characters. We could have also read it as a bunch of integers by declaring the variable `c` to be a pointer to an unsigned int rather than to an unsigned char.

But you must be thinking that those numbers don't look like a CPU instruction. Let's try to convince ourselves that those numbers are indeed the CPU instruction that implements the assignment `a = 5`. In the figure we have reprinted the value of the memory where the instruction purportedly resides:



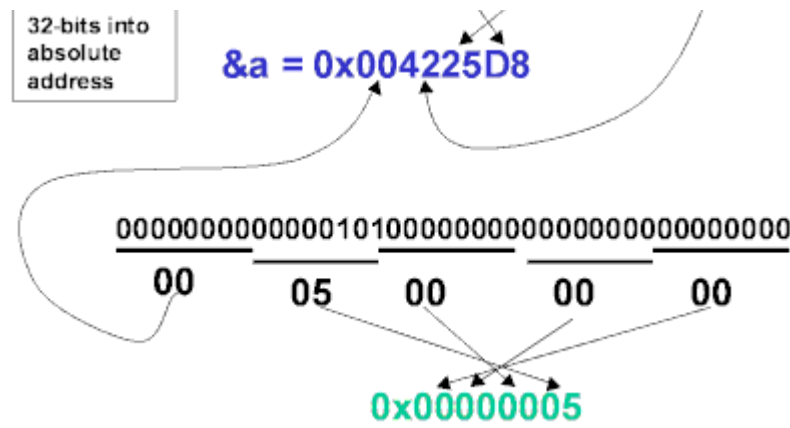


Figure 2: `a = 5` in machine code

The 80 bits are shown in two lines for clarity. Notice that a portion of the raw bits looks like the address of a. Note also that another portion of the raw bits resembles a four-byte integer of value 5. The remaining bits, 0xC705, are called the "opcode," or "operation code." These are the bits that tell the CPU what to do. 0xC705 is the Intel 486 opcode that instructs the CPU to move, into the address that follows the opcode, the four-byte literal that follows the address. Each task that the compiler might require from the CPU has to be specified through a different opcode. For example, if instead of four bytes, the compiler wanted to move only one byte (as it might if a had been declared as a character), it would have to have specified the opcode 0xC605 instead.

Opcodes are defined strictly by the manufacturer of the CPU. These opcode definitions are well publicized and standard, so that compilers can generate the correct codes. Because the opcodes for CPUs of different types are different, applications compiled for one CPU type will not work on another.

## Assembly Code

This business of CPU opcodes is usually something the compiler, and only the compiler, needs to know about. Programmers are usually happy to let the compiler map their high-level language statements into sequences of low-level CPU instructions. But this wasn't always so. Years ago, compilers were not as good as they are today, and generated code that executed much more slowly than is possible now. Programmers often preferred to manually create the critical parts of their programs directly in machine code. This was a **trade-off**: on the one hand, generating low-level instructions was very **time-consuming**; on the other, the compiler generated **slow code**.

Early in the short history of computers, programmers did not have a choice, since compilers didn't yet exist. At first, they had to manually insert the actual ones and zeros that constitute the correct CPU instructions. Using banks of physical switches, they entered sequences of digits like the one in the figure above by flipping the switch one way to represent a "one," and the other way to represent a "zero." Imagine how painstaking that must have been: programs might contain thousands of

instructions, each of which had to be entered, bit by bit, without mistakes!

To help in this nasty task, programmers developed *assemblers*. An assembler is a program that converts a mnemonic name of an instruction into the corresponding sequence of zeros and ones that represent that instruction in memory. For example, the instruction in the figure above might have a mnemonic like:

```
mov [0x4225D8], 5
```

where `mov` is short for "move," and where the **square brackets** signify that the number therein is an **address**, as opposed to an actual integer. The assembler program would translate such mnemonics into the correct sequences of bits. The assembler program itself would have to be entered through the switches the first time, bit by bit, but after that all programs, including the assembler, could be written using mnemonics.

The set of mnemonics accepted by an assembler for a specific CPU type became known as the *assembly language* for that CPU type. Because assemblers were created by the same people that made the CPU hardware, there was usually a single standard set of mnemonics for each CPU, and people started thinking of machine code (the ones and zeros) and assembly code (the mnemonics) as being one and the same. This perception is still around today even though it is no longer quite justified. The *Intel* processors, for example, to name only one type, can be programmed in at least two different assembly languages. The two languages are mapped by their respective assemblers to the exact same machine code, and they are not really that different. Still, do not be surprised if you run across different styles of assembly code.

## Assembly vs. C

Even though both assemblers and compilers translate one "language" into another, their natures are quite different. Assemblers translate one instruction at a time without analyzing or transforming the program. Notably, **assemblers** do not reorder instructions and they exercise **no choices**. For every assembly instruction presented, the assembler dumbly generates one and only one machine code instruction.

**Compilers**, on the other hand, **have choices**. They compose long sequences of instructions from source code that can be correctly implemented by many different such sequences. Compilers also analyze programs in order to optimize the result and to provide abstractions. Compilers translate the meaning of a program from the high-level to the low-level languages, whereas assemblers are mere manipulators of syntax which, to them, is meaningless.

Suppose that, instead of C and machine code, computers were programmed in English and that their CPUs only understood written Chinese. The **compiler** would be equivalent to the English/Chinese **interpreter**, and the **assembler** would be the **scribe** that wrote down the Chinese sounds uttered by the interpreter. The scribe would not have to understand a thing of what was

being said; he would only need to know which characters map to which sounds (though this is pretty hard in both Chinese and English!) The interpreter, on the other hand, has to keep the meaning in mind while completely changing the sound and structure of her sentences.

© Copyright 1999-2002 iCarnegie, Inc. All rights reserved.

## 1.5.2 Code Has Addresses, Too!

- [Code and Program Execution](#)
- [Function Calls and the Program Counter](#)
- [Function Pointers](#)

### Code and Program Execution

We now know how data and code are stored in memory. But what determines which data item or which CPU instruction will be fetched at each step of a program's execution?

We already know the answer regarding data. The address of any data required is specified explicitly in each CPU instruction that needs it. If, in C, we use a variable, the compiler will generate CPU instructions that specify the address where it has allocated that variable. Each instruction specifies the addresses of the data on which it operates.

And what determines which instruction should be fetched and executed? There is a special value kept by the CPU, called the *program counter* (often, simply *PC*), which contains the address of the instruction that is about to be executed. The CPU operates constantly in a *fetch-decode-execute* cycle, during which it:

- Fetches memory from the address contained in the program counter.
- Interprets the "opcode" just fetched and decides what the instruction means.
- Fetches operands, executes the instruction, and stores the result.

The program counter is one of several special values that are internal to the CPU and are not necessarily found in main memory. We have already seen two other such values: the stack pointer and the frame pointer. These values are special because they regulate the operation of the CPU at a very *low level*, and need to be *at hand* before anything can be made to work. Imagine the impossibility of fetching the program counter from some changing memory location: the CPU would need the program counter to find the instruction that would allow it to find the program counter!

You may have noticed that, to make any progress, the program counter needs to be modified: otherwise the CPU would be stuck always executing the same instruction. First of all, all CPU instructions automatically increment the value of the program counter, so that, in the absence of special instructions known as "branch" and "jump" instructions (more about these in a moment), the program executes instructions in memory, one after the other, in the order in which the compiler wrote them to memory. But there are CPU instructions whose purpose it is to change the value of the program counter more drastically. They are called *jump instructions* because they cause the flow of execution to "jump" to an instruction that is not next to the current one. There are also *branch instructions* that change the program counter conditionally—for example, only if an address contains a certain value.

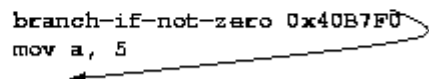
Branch instructions typically use another special CPU value called the *condition code register* (often, CCR). Instead of testing, say, whether a memory address is zero, branch instructions test whether the memory address visited by the previous instruction contained a zero. To record such a fact about the last address visited, the CPU maintains one bit that stores the truth or falsehood of the fact: if the last address contained a zero, this bit might be set to one, and to zero otherwise. The condition code register also has other such bits. For example, it might have a bit that is set to one when the last value accessed was negative. Can you think of a reason why branches are made to work this way rather than specifying directly the address they want to test, regardless of what values were accessed in the previous instruction? Think about it and try to come up with an answer.

We are now ready to see how several C code constructs might be implemented using CPU instructions. We will not use the actual instructions or their assembly mnemonics; our goal is only to give you a little insight into the task of the compiler.

```
if (a == 0) {
    a = 5;
}
```

The simple `if` statement above might translate into something like this:

Address	Instruction
0x40B7D8	fetch a
0x40B7E0	branch-if-not-zero 0x40B7F0
0x40B7E6	mov a, 5
0x40B7F0	....



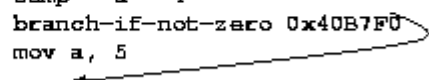
Because the first instruction *fetches* a, the branch will be taken if a is not zero. In that case, the branch instruction will change the program counter to be 0x40B7F0, causing the CPU to skip the instruction at address 0x40B7E6. That instruction, which assigns 5 to a, will only execute when a is zero.

It would seem that the CPU only allows branches that depend on a few conditions—for example, whether a variable is zero or is negative. This is true, but the compiler can get around it with a little *computation*.

```
if (a == 4) {
    a = 5;
}
```

The above statement might be translated into something like this:

Address	Instruction
0x40B7D8	temp = a - 4
0x40B7E0	branch-if-not-zero 0x40B7F0
0x40B7E6	mov a, 5
0x40B7F0	....



Now, instead of merely fetching the value we want to *test* for a branch condition, we are subtracting 4 from the value. The result is stored into

a temporary location so that it will not overwrite any variables: it is only needed to set the condition codes. This achieves the right result because testing whether  $a - 4$  is equal to zero is equivalent to testing whether  $a$  is equal to 4.

Note that if the CPU didn't use a condition code register, the branch instruction would need to specify the address of `temp`. This would result in an additional fetch. Because the CPU stores the relevant fact about `temp` (the fact that it is or it isn't zero), that fetch is not required for the branch. A reason to have branches use the condition code register is precisely that this sort of instruction sequence is extremely common in code generated by compilers.

Let's now see what might happen if we add an **else** clause. The following:

```
if (a == 4) {
    a = 5;
}
else {
    a = 4;
}
```

might become:

Address	Instruction
0x40B7D8	<code>temp = a - 4</code>
0x40B7E0	<code>branch-if-not-zero 0x40B7F6</code>
0x40B7E6	<code>mov a, 5</code>
0x40B7F0	<code>jump 0x40B800</code>
0x40B7F6	<code>mov a, 4</code>
0x40B800	<code>....</code>

Here, the red arrow represents the execution flow when  $a \neq 4$ , and the green arrow when  $a = 4$ . Here we have introduced a jump instruction, which irrevocably changes the program counter. Take a look and see how it is used to implement the ELSE clause.

CPUs don't have instructions that directly implement **loops**. Instead, loops are constructed by compilers in terms of **jumps** and **branches**. Let's look at an example which will illustrate how this is done, though we will not be covering the myriad of cases that different types of loops present. The code fragment:

```
while (a > 4) {
    a = a - 1;
}
```

might be translated into something like the following:

Address	Instruction
0x40B7D8	<code>temp = a - 4</code>
0x40B7E0	<code>branch-if-negative 0x40B800</code>
0x40B7E6	<code>temp = a - 1</code>
0x40B7F0	<code>mov a,temp</code>
0x40B7FA	<code>jump 0x40B7D8</code>
0x40B800	<code>....</code>



Try to understand why this is equivalent to the WHILE loop above.

## Function Calls and the Program Counter

When a C program calls a function, it is, in effect, making a jump to the address containing the instructions for the callee. When the function returns, the old value of the program counter is restored. This seems simple enough, but it is more subtle than it might appear at first. You should be asking yourself: where is the old program counter stored so that it can be restored when the callee returns?

You might think that the compiler can allocate any old address for this purpose, but, in doing that, the compiler would run into the same problem it has in allocating local variables. In "[1.4.2 Activation Records and Stacks](#)" we discussed how, in order to allow recursion, the compiler had to allocate local variables **in a stack of activation records**. When examining the mechanism by which the hardware managed this stack, we showed how each activation record in the stack also contained a copy of the frame pointer. The frame pointer was stored in the stack so that a different value of it could be **restored** for each function **return**.

The situation is identical with respect to the return address (the address to which program execution must return). Each function call needs to return to the address from which it was invoked, and these might all be different. It is natural to store the return address in the activation record for the callee—because there is an activation record for each function invocation, and the compiler cannot predict how many of these there will be once the program executes.

This is how it is done in practice. When making a function call, the compiler then:

- Pushes the return address (the current program counter) into the stack.
- Pushes the frame pointer into the stack.
- Sets the frame pointer equal to the stack pointer.
- Decrements the stack pointer by as many memory addresses as are required to store the local state of the callee.

Similarly, when returning from a function, the compiler:

- Sets the stack pointer equal to the frame pointer.
- Pops the value of the old frame pointer from the stack.
- Pops the value of the return address from the stack.
- Jumps to the return address.

Depending on the CPU, this might not always happen in the exact same order, but the order above is illustrative.

Because these sequences of operations are needed often, CPUs often have special instructions that do them all together. These instructions usually have mnemonics like **"jsr"** (jump to subroutine) and **"ret"** (return), though the different assembly languages may offer variations.

## Function Pointers

C doesn't provide a general way of manipulating the addresses of code, even though it provides the operators `&` and `*` to manipulate the addresses of data. However, it does provide restricted manipulation of *some* code addresses. In C, those addresses that contain the first instruction of a function can be manipulated without writing assembly code. This is done, conveniently, by applying the `&` and `*` operators to function names.

For example, a variable may be declared in the following fashion:

```
int (*newvar)(char);
```

This declaration indicates that `newvar` will be **a pointer to an object `x` of type `int X(char)`**. Since `X` is a function, `newvar` will be a pointer to a function. This is, simply, the address of the first instruction of a function.

**A pointer to a function** does not initially contain a valid value. The actual address of a function must be assigned to it before it is used. The address of a function is obtained as you might expect:

```
newvar = &existing_function;
```

In this address, the function `existing_function` will have been declared elsewhere as:

```
int existing_function (char c) {  
    }  
}
```

A function pointer may be dereferenced only by calling the function to which it points. Here is an example of an expression that would achieve that:

```
if ((*newvar)('a') == 5) {  
    .....  
}
```

**Function pointers** are useful when we need to change how an object behaves at run time. We could have a method for some object to be called through a function pointer, so that changing the function pointer would also change the method invoked. This is, in fact, how some versions of C++ implement C++ objects in terms of C constructs.



## 1.5.3 Registers

- The CPU Also Has Memory
- Spying on Registers
- Register Allocation and Compiler Optimization

### The CPU Also Has Memory

In the discussion so far we have talked about how data and instructions are stored in the computer's memory. We have presented execution as a process in which instructions and data are repeatedly fetched from memory under the direction of the instructions themselves. In this picture, the CPU has been a fast but simple participant that carries out simple instructions and remembers little or nothing from one instruction to the next.

This is not strictly true. The CPU also maintains its own banks of memory. These memories cannot be directly manipulated from the programming language (for example, C). Instead, they speed up the program's execution or simplify the hardware design. They usually do this by:

- Providing a shorthand to refer to data that is currently in use.
- Keeping data handy when it is needed repeatedly.


The "shorthand" is often in the form of a small range of integers. For instance, if a value is allocated by the compiler to address 0x00123456 in real memory, the value could be temporarily stored in a small CPU memory and given a commensurately small address—for example, 0x3—for the duration of its temporary existence.

Instructions that specified 0x3 rather than 0x00123456 when referring to that value would be smaller and could then be fetched more quickly.

To distinguish such CPU memory addresses from the normal addresses, assemblers usually provide different

syntax for them. In many assembly languages, for instance, the CPU memory address 0x3 would be written as r3 in assembly. In the common Intel assembly language syntax, the CPU memory addresses have wholly symbolic names, so that CPU memory location 0x3 might be indicated with something like EAX or EBX.

The idea of **keeping the data handy** is a very common one, in computer technology as well as in life in general. The idea is that we temporarily put the things we will need soon in a place that is easily accessible rather than keeping everything on one large, uniform, storeroom. We don't pay for lunch by repeatedly traveling between the bank and the restaurant, delivering one dollar bill at a time. We carry several dollar bills in our wallet instead. We might also not carry all of our dollar bills in the wallet, because, lucky us, we might have more money in the bank than could fit in our wallet. Computers do the same: they have **fast, accessible, but small**, memories where they store values that will be needed soon or needed repeatedly. But, no matter what, all values are also kept in one large (and relatively slow) memory with capacity to store everything a program will ever need.

In any case, the values that the CPU keeps in its small memories are surrogates  for the values that are stored in the real, big, main memory. This has to be so because, being small, the CPU memories are in high demand and cannot store any value permanently. This has profound implications. One is that we must make sure not to access the value in main memory while its surrogate is in one of the CPU memories, because the value in main memory might be obsolete or because doing so might make the surrogate obsolete. To see how this might happen, consider the situation in which you and your spouse have two separate checkbooks for one single bank account: when you write a check based on your checkbook balance, how do you know that your spouse hasn't just spent most of the money buying you a very

expensive anniversary gift? The problem is that because the two copies of the balance can be updated separately, either copy may be outdated.

The fact that the values temporarily stored in CPU memories are surrogates for values in normal locations in main memory has other implications. You might have already questioned **who** or **what** decides **when** to make a surrogate copy of a data item and **how long** the surrogate should exist. This is an excellent question. Having small CPU memories adds substantially to the complexity of managing memory, but these memories speed things up so much that they are worth the trouble. Let's partially answer the question: Who makes and destroys surrogates? In today's computers there are two answers:

- **The Compiler:** Some of the CPU memories can be explicitly managed by the compiler. This means that the compiler decides which portions of the data, and when, should be moved from main memory to the CPU memory and vice versa. The compiler does the actual data moving by using CPU instructions that are available for this specific purpose. The compiler also **ensures** that the data in main memory is not accessed while a surrogate exists in a CPU memory. If the compiler does this intelligently, the program will execute a lot faster. This is difficult, but over the years compilers have been getting very good at it. Some say that compilers are now better than most people.

The CPU memory that is managed explicitly by the compiler is usually called a **register bank**, and the individual memory cells that store each data item are called **registers**. CPUs typically have several registers, but not a large number of them. The Intel architectures have somewhere between 6 and 16 registers, depending how you count. Other architectures often have more registers, but not

that many more.

- **The Hardware:** Some of the CPU memories are managed by the hardware. The hardware can be **faster** than the compiler in moving data back and forth, but it has no knowledge of what the program is likely to do with the data in the near future. So the hardware can **create and destroy surrogates faster** than the compiler, but it often creates surrogates for the wrong things. It is nevertheless so useful that most computers have such hardware-managed memories inside the CPU.

These memories are typically called ***caches***, because they **automatically** (that is, without instruction from the compiler) “cache” data that may be about to be re-used. Caches vary widely in size, but it is not unusual for them to be able to hold several thousand data items at once. We will talk more about caches in “[Unit 5. Memory Operation and Performance](#).”


## Spying on Registers

It’s about time we take a look at how the compiler uses registers. [Download](#) the following code and load it into Visual C++.

```
int a;
int b = 4;
int c = 3;

int main ( int argc, char * argv[])
{
    a = b + 3 * c;
    return 0;
}
```

Place a breakpoint in the line `a = b + 3 * c` and press F5 to run the debugger. Wait until the debugger stops at the breakpoint.

Let's look at the instructions that the compiler generated for us. You can do this by clicking on the  icon in the **Debug** toolbar, or by selecting the **Debug Windows::Disassembly** item from the **View** menu, or by pressing **Alt+8**. The disassembly window that appears is rather complex at first sight, but don't be intimidated. The first column of this window contains the **addresses** that contain the instructions for the program. To the right of the addresses Visual C++ usually shows the instructions as they are **encoded into byte values**, though your particular settings may not show this. To the right of the byte values are shown the assembly mnemonics for the instructions that are at the indicated addresses. You can toggle the display of the byte values by checking and unchecking the **Code Bytes** flag in the menu that appears when you right-click on the disassembly window, as shown in the figure:

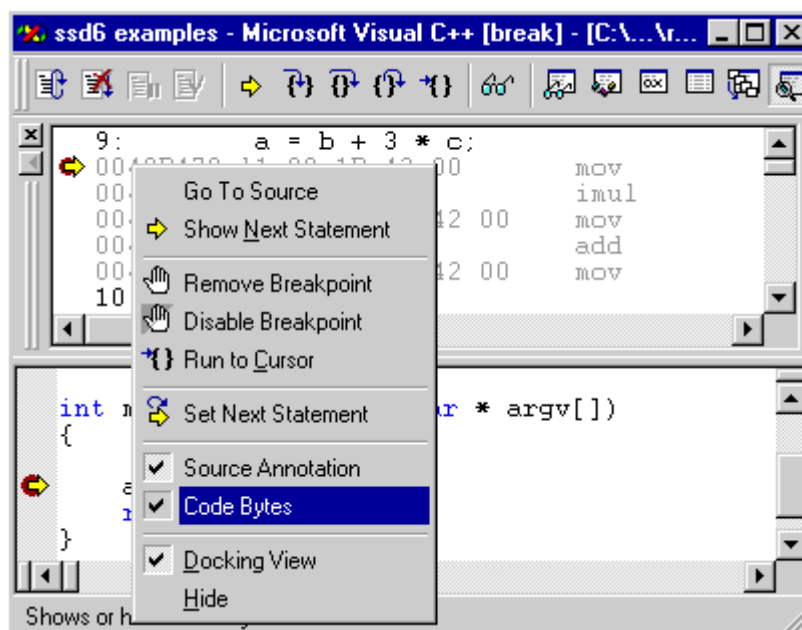


Figure 1: Toggling the display of the byte values

The assembly mnemonics probably don't make a lot of sense to you. Neither do they to us! However, there are some things to be learned from them even without trying to understand absolutely everything you see. Let's look at what the disassembly window shows, which should be



something similar to the following:

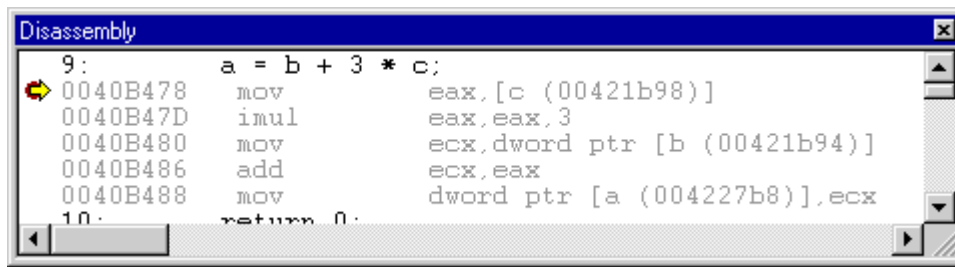


Figure 2: Disassembly of  $a = b + 3 * c$

The first thing to notice is that Visual C++ has helpfully **annotated the assembly code with the original C source statements** that correspond to each group of instructions. In the disassembly window we see a line like 9:  $a = b + 3 * c$ ; right before the CPU instructions that implement line 9 in the original source code. The source line is *not* part of the assembly code; it is printed there merely for our convenience. If it does not appear in your window, you may have to check the **Source Annotation** flag in the right-click menu.

Another thing to notice is that Visual C++ shows the usual red circle and yellow arrow to indicate that execution is currently stopped at a specific instruction. If you stop execution from the source window, the yellow arrow will move several instructions down, until it reaches the first instruction of several that implement the next line of C source code. If you stop execution from the disassembly window, however, Visual C++ will **only step over one assembly instruction**. Try it by pressing F10 while the mouse pointer is in the disassembly window. At this point, the C statement  $a = b + 3 * c$  has started executing, but has not yet finished.

Now, let's look at the assembly code.

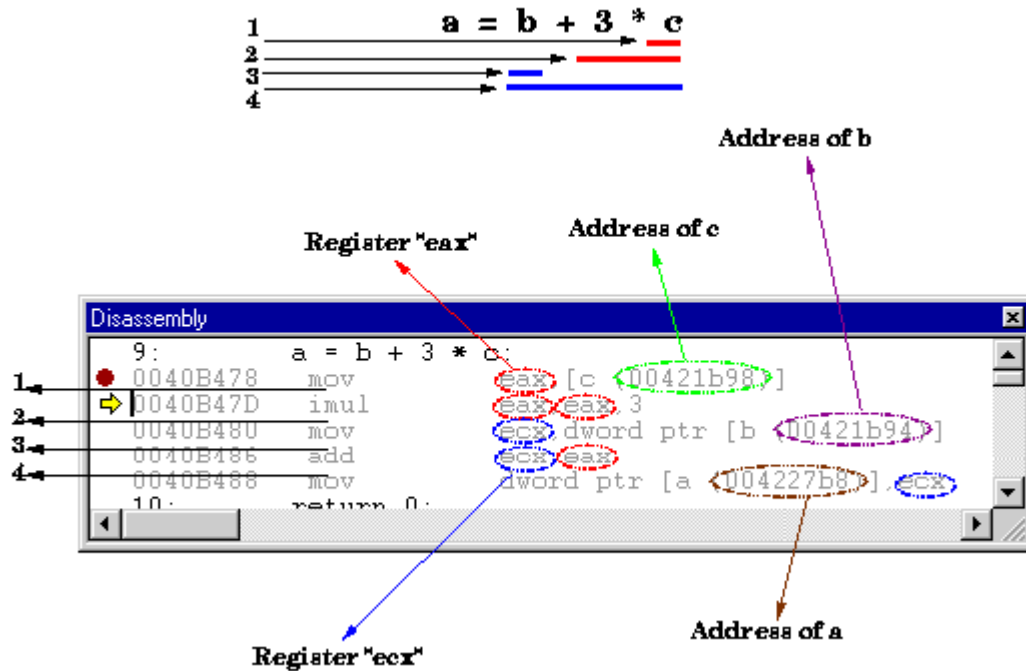
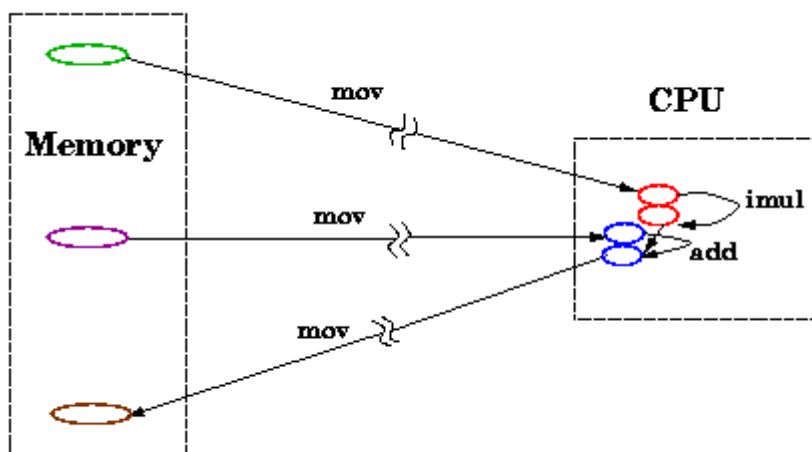


Figure 3: Disassembly of  $a = b + 3 * c$

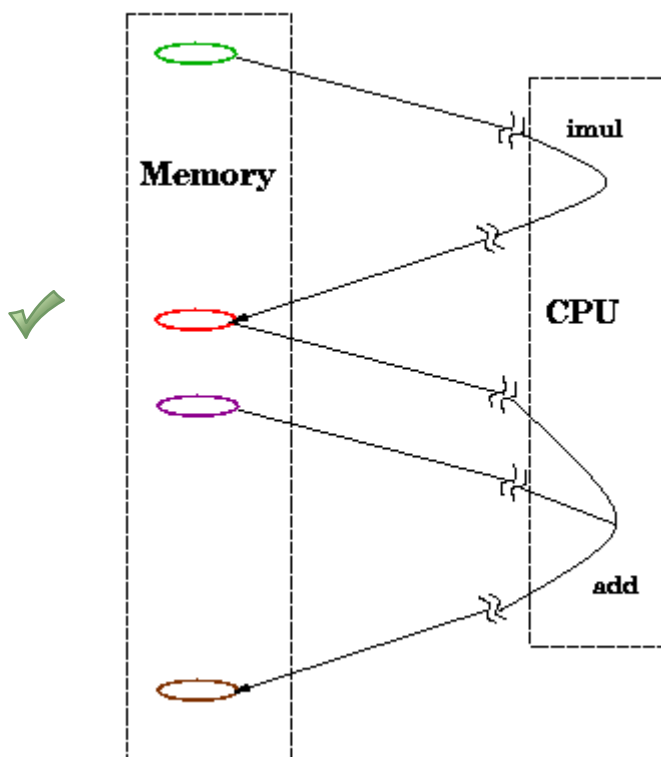
Five instructions implement the C statement  $a = b + 3 * c$ . There are three "move" instructions, one add instruction, and one integer-multiply instruction. Perhaps it will be simpler to look at a graphic representation of this execution. In the figure below, distance is meant to represent the time that it takes an instruction to execute. Instructions that only need data already resident in the CPU execute extremely fast, whereas instructions that require access to memory execute relatively slowly. This is mostly because memory is large and inexpensive.



$$a = b + 3 * c$$

Figure 4: Execution graph for  $a = b + 3 * c$

Let's consider what instructions the compiler might generate if the CPU didn't have any registers. First, it would need to allocate another variable in memory to hold the temporary value  $3 * c$ . It would then need to execute things in a fashion similar to the one shown in the figure:



$$a = b + 3 * c$$
 in a CPU with no registers">

Figure 5: Execution graph for  $a = b + 3 * c$  in a CPU with no registers

There are at least two things to note about this graph as it contrasts with the other. The first is that the graph is a lot **longer** and there are **more exchanges** between memory and the CPU. This will make the program execute much **more slowly**. The second noteworthy feature is that the new graph is implemented with only two instructions rather than the previous five. So, two

instructions can take longer than five! Not all instructions do the same amount of work.

## Register Allocation and Compiler Optimization

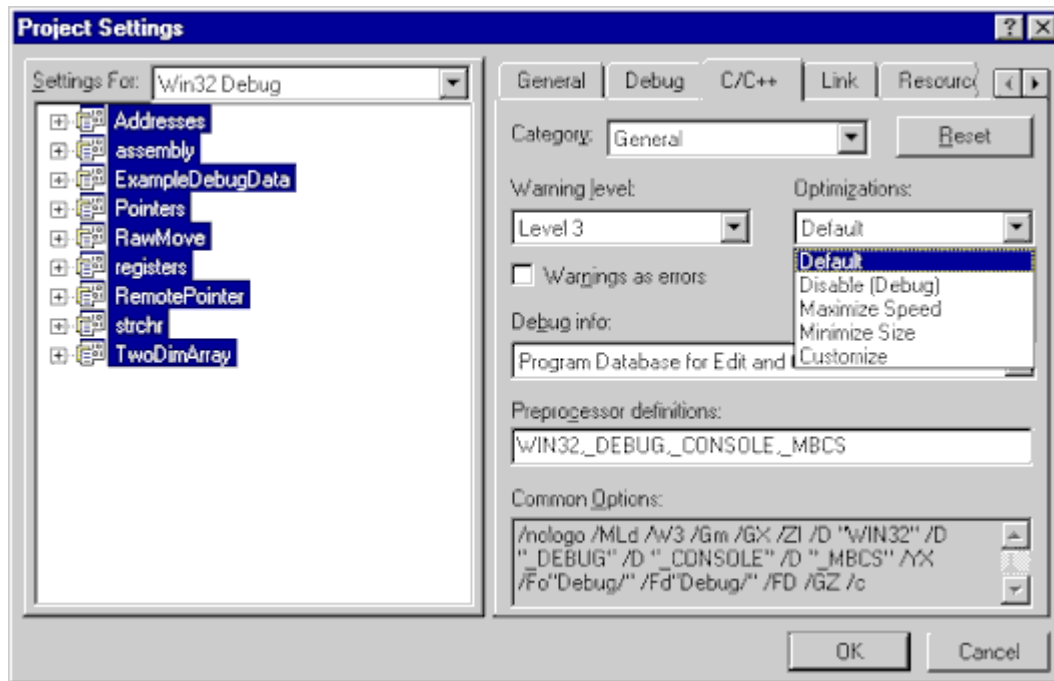
The compiler would like to be able to allocate a register for every variable in the source program because every variable that lives in a register speeds up the program's execution by some amount. But CPUs only have a few registers, so the compiler needs to decide when and how to use each register as a surrogate for which variable. This task, called *register allocation*, is extremely difficult. It is so difficult that compilers do not try to allocate registers in the best way possible: it would take too long. Instead, they use rules of thumb to allocate registers to variables in ways that typically result in pretty good, though not the best, performance.

The extent to which compilers optimize register allocation depends on the *optimization level* that they are told to use. Higher optimization levels result in more aggressive optimizations which take longer to compile, whereas lower optimization levels result in slow execution. Compilers do not always use their most aggressive optimizations for two reasons:

- Compilation with aggressive optimizations takes a long time.
- Aggressive optimizations change the code substantially: It may be difficult, both for people and for debuggers, to relate the resulting machine code to the original source code. Debugging, in particular, becomes almost impossible at high optimization levels.

For these reasons, compilers are most often run at low optimization levels during development, when speed of compilation and debugging are important. Once the programs are working, they are compiled one final time with the aggressive optimizations turned on, because

after that point, speed of execution is important and debugging is no longer necessary (supposedly). Visual C++ allows us to change the optimization level in the C/C++ tab of the **Project::Settings** menu item.



Visual

C++">

Figure 6: Changing the optimization level in Visual C++

The optimization level affects a lot of things besides register allocation. Some programs may become quite unrecognizable when compiled with high optimization. Give it a try!

© Copyright 1999–2002 iCarnegie, Inc. All rights reserved.