**The Science of Building a Compiler**

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

**Modeling in Compiler Design and Implementation**

 The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

Some of most fundamental models are finite-state machines and regular expressions, which we shall meet in Chapter 3. These models are useful for de-scribing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to de-scribe the syntactic structure of programming languages such as the nesting of parentheses or control constructs. We shall study grammars in Chapter 4. Sim-ilarly, trees are an important model for representing the structure of programs and their translation into object code, as we shall see in Chapter 5.

**The Science of Code Optimization**

 The term "optimization" in compiler design refers to the attempts that a com-piler makes to produce code that is more efficient than the obvious code. "Op-timization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively par-allel computers require substantial optimization, or their performance suffers by orders of magnitude. With the likely prevalence of multicore machines (com-puters with chips that have large numbers of processors on them), all compilers will have to face the problem of taking advantage of multiprocessor machines.

It is hard, if not impossible, to build a robust compiler out of "hacks." Thus, an extensive and useful theory has been built up around the problem of optimizing code. The use of a rigorous mathematical foundation allows us to show that an optimization is correct and that it produces the desirable effect for all possible inputs. We shall see, starting in Chapter 9, how models such as graphs, matrices, and linear programs are necessary if the compiler is to produce well optimized code.

On the other hand, pure theory alone is insufficient. Like many real-world problems, there are no perfect answers. In fact, most of the questions that we ask in compiler optimization are undecidable. One of the most important skills in compiler design is the ability to formulate the right problem to solve. We need a good understanding of the behavior of programs to start with and thorough experimentation and evaluation to validate our intuitions.

Compiler optimizations must meet the following design objectives:

        The optimization must be correct, that is, preserve the meaning of the compiled program,

 •  The optimization must improve the performance of many programs,

 •  The compilation time must be kept reasonable, and

 •  The engineering effort required must be manageable.

It is impossible to overemphasize the importance of correctness. It is trivial to write a compiler that generates fast code if the generated code need not be correct! Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct.

The second goal is that the compiler must be effective in improving the performance of many input programs. Normally, performance means the speed of the program execution. Especially in embedded applications, we may also wish to minimize the size of the generated code. And in the case of mobile devices, it is also desirable that the code minimizes power consumption. Typically, the same optimizations that speed up execution time also conserve power. Besides performance, usability aspects such as error reporting and debugging are also important.

Third, we need to keep the compilation time short to support a rapid devel-opment and debugging cycle. This requirement has become easier to meet as machines get faster. Often, a program is first developed and debugged without program optimizations. Not only is the compilation time reduced, but more importantly, unoptimized programs are easier to debug, because the optimiza-tions introduced by a compiler often obscure the relationship between the source code and the object code. Turning on optimizations in the compiler sometimes exposes new problems in the source program; thus testing must again be per-formed on the optimized code. The need for additional testing sometimes deters the use of optimizations in applications, especially if their performance is not critical.

Finally, a compiler is a complex system; we must keep the system sim-ple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimenta-tion. We normally start by formulating the problem based on our intuitions on what the important issues are.

### **Applications of compiler technology:**

#### **1. Implementation of High-level Programming**

*A high-level programming language defines a programming abstraction*: the programmer specifies an algorithm in the language, and the compiler must translate it to the target language. Higher-level programming languages are sometimes easier to develop in, but they are inefficient, therefore the target applications run slower. Low-level language programmers have more control over their computations and, in principle, can design more efficient code. Lower-level programs, on the other hand, are more difficult to build and much more difficult to maintain. They are less portable, more prone to errors, and more complex to manage. Optimized compilers employ ways to improve the performance of generated code, compensating for the inefficiency of high-level abstractions.

In actuality, programs that utilize the register keyword may lose efficiency since programmers aren’t always the best judges of extremely low-level matters like register allocation. The ideal register allocation approach is very reliant on the design of the machine. Hardwiring low-level resource management decisions like register allocation may actually harm performance, especially if the application is executed on machines that aren’t meant for it.

#### 2. **Optimization of computer architectures**

Aside from the rapid evolution of computer architectures, there is a never-ending demand for new compiler technology. Almost all high-performance computers leverage parallelism and memory hierarchies as essential methods. Parallelism may be found at two levels: at the instruction level, where many operations are performed at the same time, and at the processor level, where distinct threads of the same program are executed on different processors. Memory hierarchies address the fundamental problem of being able to produce either extremely fast storage or extremely huge storage, but not both.

#### 3**. Design of new computer architectures**

In the early days of computer architecture design, compilers were created after the machines were built. That isn’t the case now. Because high-level programming is the norm, the performance of a computer system is determined not just by its sheer speed, but also by how well compilers can use its capabilities. Compilers are created at the processor-design stage of contemporary computer architecture development, and the resultant code is used to evaluate the proposed architectural features using simulators.

#### 4**. Program Translations:**

The compilation is typically thought of as a translation from a high-level language to the machine level, but the same approach may be used to translate across several languages. The following are some of the most common applications of software translation technologies.

* Compiled Simulation
* Binary translation
* Hardware Syntheses
* Database Query Interpreters

#### 5. **Software productivity tools**

Programs are possibly the most complex technical objects ever created; they are made up of a plethora of little elements, each of which must be accurate before the program can function properly. As a result, software mistakes are common; errors can cause a system to crash, generate incorrect results, expose a system to security threats, or even cause catastrophic failures in key systems. Testing is the most common method for discovering program flaws.

A fascinating and interesting complementary option is the use of data-flow analysis to statically discover problems (that is before the program is run). Unlike program testing, the data-flow analysis may uncover vulnerabilities along any possible execution path, not only those used by the input data sets. Many data-flow-analysis techniques, originally developed for compiler optimizations, may be used to build tools that assist programmers with their software engineering responsibilities.

#### 6**. Artificial Intelligence:**

 Compilers are used in the field of artificial intelligence (AI) to optimize and generate code for deep learning models, computer vision, natural language processing, and other AI applications. AI compilers can optimize code for specific hardware architectures and can generate highly efficient code for AI workloads.

#### 7.**Gaming:**

Game development often involves the use of compilers to generate code that runs on game consoles and PCs. Gaming compilers are optimized for performance, allowing game developers to create immersive, high-performance games.

#### 8.**Security:**

Compilers are used in security applications to create code that is resistant to various forms of attacks, including buffer overflows, code injections, and other security vulnerabilities. Security compilers can generate code that is highly resistant to reverse engineering and tampering.

#### 9. **Embedded Systems:**

Embedded systems are computer systems that are designed to perform specific functions in various devices such as automobiles, medical equipment, and consumer electronics. Compilers are used to generate machine code that runs on these devices, ensuring efficient use of resources and optimal performance.

#### 10. **High-Performance Computing:**

 Compilers are essential in the field of high-performance computing (HPC), which involves using supercomputers or clusters of computers to perform complex computations. HPC compilers are optimized for parallel processing and can generate highly efficient code for scientific simulations, data analysis, and other computationally intensive applications.

## **Programming Language Basics**

To design an efficient compiler, we should know some language basics.

Programming Language Basics

## **Static and Dynamic Distinction**

* **Static** - Events occur at compile time.
* **Dynamic** - Events occur at run time.

## **Example**

* The scope of a declaration of x is the region of the program in which uses of x refer to this declaration.
* A language uses static scope or lexical scope if it is possible to determine the scope of a declaration by looking only at the program.
* Otherwise, the language uses dynamic scope. With dynamic scope, as the program runs, the same use of x could refer to any of several different declarations of x.

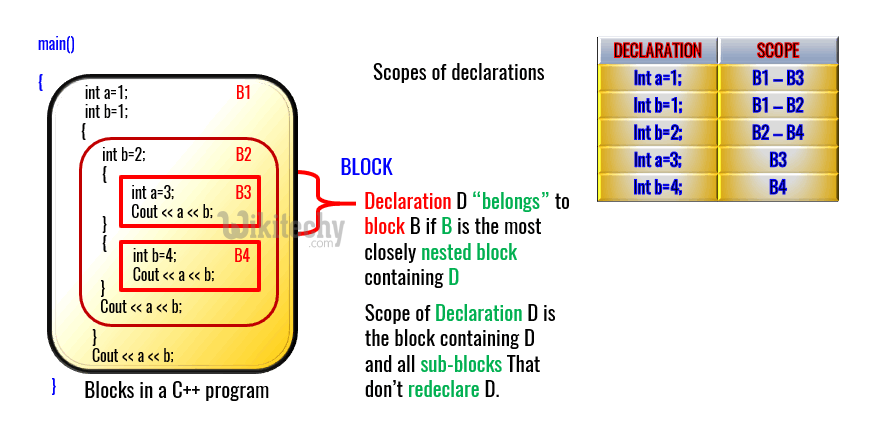
## **Environment and States**

* The environment is mapping from names to locations in the store. Since [variables](https://www.wikitechy.com/tutorials/javascript/static-variable-in-javascript) refer to locations, we could alternatively define an environment as a mapping from names to variables.
* The state is a mapping from locations in store to their values.

Environment and States

## **Static Scope and Block Structure**

* The scope rules for [C](https://www.wikitechy.com/tutorials/c-programming/) are based on program structure. The scope of a declaration is determined implicitly by where the declaration appears in the program.
* Programming languages such as [C++](https://www.wikitechy.com/tutorials/c++/), [Java](https://www.wikitechy.com/tutorials/java/), and [C#](https://www.wikitechy.com/tutorials/csharp/), also provide explicit control over scopes through the use of keywords like public, private, and protected.
* A block is a grouping of declarations and statements. C uses braces { and } to delimit a block, the alternative use of begin and end in some languages.



Static Scope and Block Structure

## **Explict Access Control**

* [Classes](https://www.wikitechy.com/tutorials/c++/c++-classes-and-objects)and [structures](https://www.wikitechy.com/tutorials/c++/c++-structure-and-function) introduce a new scope for their members.
* If p is an object of a class with a field (member) x, then the use of x in p.x refers to field x in the class definition.
* Through the use of keywords like public, private, and protected, object oriented languages such as C++ or Java provide explicit control over access to member names in a super class. These keywords support encapsulation by restricting access.
  + **Public** - Public names are accessible from outside the class
  + **Private** - Private names include method declarations and definitions associated with that class and any "friend" classes.
  + **Protected** - Protected names are accessible to subclasses.

## **Dynamic Scope**

* The term dynamic scope, however, usually refers to the following policy: a use of a name x refers to the declaration of x in the most recently called procedure with such a declaration.
* Dynamic scoping of this type appears only in special situations. The two dynamic policies are:
  + Macro expansion in the [C preprocessor](https://www.wikitechy.com/tutorials/c-programming/preprocessor-directives-in-c)
  + Method resolution in [object-oriented programming](https://www.wikitechy.com/tutorials/python/python-object-oriented-programming).

Since dynamic scoping is very uncommon in the familiar languages, we consider the following code as our example.

## **Sample Code**

int x = 10;

*// Called by g()*

int f()

{

return x;

}

*// g() has its own variable*

*// named as x and calls f()*

int g()

{

int x = 20;

return f();

}

main()

{

printf(g());

}

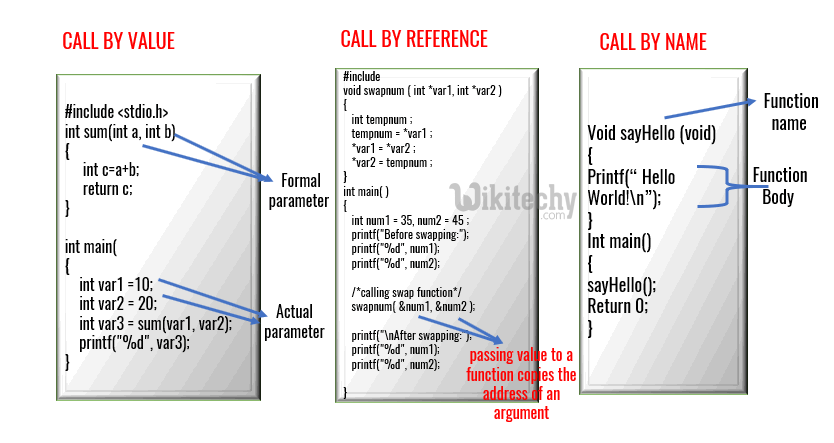
Copy Code

## **Output**

20

## **Parameter Passing Mechanisms**

* Every language has some method for passing parameters to functions and procedures.
* **Formal Parameters:** The identifier used in a method to stand for the value that is passed into the method by a caller.
* **Actual Parameters:** The actual value that is passed into the method by a caller.
  + **Call by Value** - The actual parameter is evaluated (if it is an expression) or copied (if it is a variable) in a formal parameter.
  + **Call by Reference** - The address of the actual parameter is passed as value of the corresponding formal parameter.
  + **Call by Name** - The Called object execute as if the actual parameter were substituted literally for the formal parameter.



Formal and Actual Parameter

## **Aliasing**

* When two names refer to the same location in memory.
* There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value.
* It is possible that two formal parameters can refer to the same location; such variables are said to be aliases of one another.
* As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other.

