**Variable Declaration, Definition and Scope**

**Variable:** A variable in simple terms is a storage place, which has some memory allocated to it. A variable is used to store some form of data. Different types of variables require different amounts of memory, and have some specific set of operations, which can be applied on them.

# Variable Declaration

type variable\_name; // single variable

type variable1\_name, variable2\_name, variable3\_name; // multiple variables

* A variable name can consist of alphabets (both upper and lower case), numbers and the underscore ‘\_’ character
* A variable name must not start with a number
* ANSI C++ has no limits on the length of a name
* Reserved identifiers cannot be used as names for the program variable
* If a variable is shared by more than one file, some OS impose a restriction on the length of such a variable name
* Variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking

# Why variable name does not start with numbers in C?

The compiler has 7 phases as following:

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Code Generation
7. Symbol Table

Backtracking is avoided in lexical analysis phase while compiling the piece of code.

When you are parsing the token, you only have to look at the first character to determine if it is an identifier or literal and then send it to the correct function for processing. Therefore, that is a performance optimization.

The variable like Apple; , the compiler will know it’s a identifier right away when it meets letter ‘A’ character in the lexical Analysis phase.

However, a variable like 123apple; , compiler won’t be able to decide if it’s a number or identifier until it hits ‘a’ and it needs backtracking to go in the lexical analysis phase to identify that it is a variable. However, it is not supported in compiler.

# Variable Definition

Variable definition is the part where the variable is assigned a memory location and a value.

Most of the times, variable declaration and definition are done together.

char ch = ‘a’; // declaration and definition

float b; // declaration and definition

// b is allocated memory and assigned some garbage value

*A variable that has not been given a known value (usually through initialization or assignment) is called an uninitialized variable.*

*Using the value from an uninitialized variable is undefined behavior.*

**Is it possible to have separate declaration and definition?**

Yes, for only declaration of a variable use keyword extern

extern int y;

# Lvalues and Rvalues

## Lvalue

Expression that refer to a memory location is called “lvalue” expression

An lvalue may appear as either the left-hand or right-hand side of an assignment

Variables are lvalues

## Rvalue

Rvalue refers to a data value that is stored at some address in memory

Rvalue is an expression that cannot have a value assigned to it, which means an rvalue may appear on the right but not left hand side of an assignment

Numeric literals are rvalues

# Dynamic initialization of variables

C++ permits initialization of variables at run time:

Float area = 3.14 \* rad \* rad;

Dynamic initialization is extensively used in OOP. We can create exactly the type of object needed, using information that is known only at run time.

# Re-declaration of global variable in C

Consider the following programs:

|  |  |
| --- | --- |
| #include <stdio.h>  int main() {  int x;  int x = 5; // CE  printf("%d", x);  return 0;  }  // In C, CE: redeclaration of ‘x’ with no linkage  // In C++, CE: redeclaration of ‘int x’ | #include <stdio.h>  int x;  int x = 5; // CE, C++  int main() {  printf("%d", x);  return 0;  }  In C, Output: 5  In C++, CE: redefinition of ‘int x’ |
| #include <stdio.h>  int x = 5;  int x; // CE, C++  int main() {  printf("%d", x);  return 0;  }  In C, Output: 5  In C++, CE: redefinition of ‘int x’ | #include <stdio.h>  int x;  int x = 5; // CE, C++  int x;  int main() {  printf("%d", x);  return 0;  }  In C, Output: 5  In C++, CE: redefinition of ‘int x’ |
| #include <stdio.h>  int x = 10;  int x = 5;  int main() {  printf("%d", x);  return 0;  }  // In C, CE: redefinition of ‘x’  // In C++, CE: redefinition of ‘int x’ |  |

In C, **C allows a global variable to be declared again when first declaration doesn’t initialize the variable.**

Redeclaration or definition of local variables is not allowed.

In C++, Redeclaration or definition of global variable is not allowed

# How Linkers Resolve Global Symbols Defined at Multiple Places?

At compile time, the compiler exports each global symbol to the assembler as either strong or weak, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. Functions and initialized global variables get strong symbols. Uninitialized global variables get weak symbols.

Given this notion of strong and weak symbols, Unix linkers use the following rules for dealing with multiple defined symbols:

Rule 1: Multiple strong symbols (with same variable name) are not allowed.

Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol.

Rule 3: Given multiple weak symbols, choose any of the weak symbols.

For example, suppose we attempt to compile and link the following two C modules:

/\* foo1.c \*/

int main() { return 0; }

/\* bar1.c \*/

int main() { return 0; }

In this case, the linker will generate an error message because the strong symbol main is defined multiple times (rule 1).

/\* foo3.c \*/

#include <stdio.h>

void f(void);

int x = 15213;

int main() { f(); printf("x = %d\n", x); return 0; }

/\* bar3.c \*/

int x;

void f() { x = 15212; }

If x is uninitialized in one module, then the linker will quietly choose the strong symbol defined in the other (rule 2)

/\*a.c\*/

#include <stdio.h>

void b(void);

int x;

int main() { x = 2016; b(); printf("x = %d ",x); return 0; }

/\*b.c\*/

#include <stdio.h>

int x;

void b() { x = 2017; }

If there are two weak definitions of x, , then the linker will quietly choose any weak symbol (rule 3)

NOTE: (VERY IMPORTANT)

The application of rules 2 and 3 can introduce some insidious run-time bugs that are incomprehensible to the unwary programmer, especially if the duplicate symbol definitions have different types.

Example : “x” is defined as an int in one module and a double in another.

/\*a.c\*/

#include <stdio.h>

void b(void);

int x = 2016;

int y = 2017;

int main() { b(); printf("x = 0x%x y = 0x%x \n", x, y); return 0; }

/\*b.c\*/

double x;

void b() { x = -0.0; }

Output:

x = 0x0 y = 0x80000000

This is a subtle and nasty bug, especially because it occurs silently, with no warning from the compilation system, and because it typically manifests itself much later in the execution of the program, far away from where the error occurred. In a large system with hundreds of modules, a bug of this kind is extremely hard to fix, especially because many programmers are not aware of how linkers work. When in doubt, invoke the linker with a flag such as the **gcc -fno-common flag**, which triggers an error if it encounters multiple defined global symbols.

# Types of variables

Three types of variables based on the scope of variables in C++:

1. Local Variables
2. Instance Variables
3. Static Variables

## Local Variables

* A variable defined within a block or method or constructor is called local variable.
* These variables are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
* The scope of these variables exists only within the block in which the variable is declared. I.e. we can access these variables only within that block.
* Initialization of Local Variable is Mandatory.

## Instance Variables

* Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
* As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
* Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
* Initialization of Instance Variable is not mandatory.
* Instance Variable can be accessed only by creating objects.

## Static Variables

* Static variables are also known as Class variables.
* These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
* Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
* Static variables are created at the start of program execution and destroyed automatically when execution ends.
* Initialization of Static Variable is not mandatory. Its default value is 0
* If we access the static variable like Instance variable (through an object), the compiler will show the warning message and it won’t halt the program. The compiler will replace the object name to class name automatically.

## Instance variable Vs Static variable

Syntax for static and instance variables:

class Example {

static int a; // static variable

int b; // instance variable

}

Each object will have its own copy of instance variable whereas We can only have one copy of a static variable per class irrespective of how many objects we create.

Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of instance variable. In case of static, changes will be reflected in other objects as static variables are common to all object of a class.

We can access instance variables through object references and Static Variables can be accessed directly using class name.

# How to access global variable if there is a local variable with same name in C/ C++

In C using extern

|  |  |  |
| --- | --- | --- |
| In C using extern |  | In C++ using scope resolution operator (::) |
| #include <stdio.h>  int x = 50; // Global x  int main() {  int x = 10; // Local x  {  extern int x;  printf("Value of global x is %d\n", x);  }  printf("Value of local x is %d\n", x);  return 0;  }  Output:  Value of global x is 50  Value of local x is 10 |  | #include <iostream>  using namespace std;  int x = 50; // Global x  int main() {  int x = 10; // Local x  cout << "Value of global x is " << ::x << endl;  cout << "Value of local x is " << x;  return 0;  }  Output:  Value of global x is 50  Value of local x is 10 |

# How are variables scoped in C – Lexical (Static) or Dynamic?

In C, variables are always **statically (or lexically) scoped** i.e., binding of a variable can be determined by program text and is independent of the run-time function call stack.

# include <stdio.h>

int x = 0;

int f() { return x; }

int g() {

int x = 1;

return f();

}

int main() {

printf("%d", g());

printf("\n");

return 0;

}

Output:

0 // In C & C++

# Scope rules in C

Scope of an identifier is the part of the program where the identifier may directly be accessible. In C, all identifiers are lexically(or statically) scoped.

C scope rules can be covered under the following categories:

|  |  |
| --- | --- |
| SCOPE | MEANING |
| File Scope  (Global Scope) | Scope of a Identifier starts at the beginning of the file and ends at the end of the file. It refers to only those Identifiers that are declared outside of all functions. The Identifiers of File scope are visible all over the file Identifiers having file scope are global.  Note: To restrict access to the current file only, global variables can be marked as static. |
| Block Scope | Scope of a Identifier begins at opening of the block / ‘{‘ and ends at the end of the block / ‘}’. Identifiers with block scope are local to their block |
| Function Prototype Scope | Identifiers declared in function prototype are visible within the prototype.  This scope does not include the function definition, but just the function prototype. |
| Function scope | Function scope begins at the opening of the function and ends with the closing of it. Function scope is applicable to labels only. A label declared is used as a target to goto statement and both goto and label statement must be in same function |

## Variable shadowing (name hiding)

Each block defines its own scope region. So what happens when we have a variable inside a nested block that has the same name as a variable in an outer block? When this happens, the nested variable “hides” the outer variable in areas where they are both in scope. This is called name hiding or shadowing.

**What about functions and parameters passed to functions?**

A function itself is a block. Parameters and other local variables of a function follow the same block scope rules.

**Can variables of the block be accessed in another subsequent block?**

No, a variable declared in a block can only be accessed inside the block and all inner blocks of this block.

**Can we access global variable if there is a local variable with same name?**

Check [How to access global variable if there is a local variable with same name in C/ C++](#_How_to_access)

# Initialization of global and static variables in C and C++

In C, the compiler itself initializes static and global variables. Therefore, they must be initialized with a constant value.

#include <stdio.h>

#include <stdlib.h>

int main(void) {

static int \*p = (int\*)malloc(sizeof(p));

\*p = 10;

printf("%d", \*p);

return 0;

}

// In C, CE: initializer element is not constant

// In C++, 10

#include <stdio.h>

#include <stdlib.h>

int \*p = (int\*)malloc(sizeof(p));

int main(void) {

\*p = 10;

printf("%d", \*p);

return 0;

}

// In C, CE: initializer element is not constant

// In C++, 10

#include <stdio.h>

int fun(int x) {

return (x+5);

}

int y = fun(20);

int main() {

printf("%d ", y);

return 0;

}

// In C, CE: initializer element is not constant

// In C++, 25

# Internal Linkage and External Linkage in C

(VERY IMPORTANT)

It is often quite hard to distinguish between scope and linkage, and the roles they play. This article focuses on scope and linkage, and how they are used in C language.

Note: All C programs have been compiled on 64 bit GCC 4.9.2. Also, the terms “identifier” and “name” have been used interchangeably in this article.

**Scope:** Scope of an identifier is the part of the program where the identifier may directly be accessible. In C, all identifiers are lexically (or statically) scoped.

**Linkage:** Linkage describes how names can or cannot refer to the same entity throughout the whole program or one single translation unit.

The above sounds similar to Scope, but it is not so. To understand what the above means, let us dig deeper into the compilation process.

**Translation Unit:** A translation unit is a file containing source code, header files and other dependencies. All of these sources are grouped together in a file for they are used to produce one single executable object. It is important to link the sources together in a meaningful way. For example, the compiler should know that printf definition lies in stdio header file.

In C and C++, a program that consists of multiple source code files is compiled one at a time. Until the compilation process, a variable can be described by it’s scope. It is only when the linking process starts, that linkage property comes into play. Thus, scope is a property handled by compiler, whereas linkage is a property handled by linker.

The Linker links the resources together in the linking stage of compilation process. The Linker is a program that takes multiple machine code files as input, and produces an executable object code. It resolves symbols (i.e, fetches definition of symbols such as “+” etc..) and arranges objects in address space.

Linkage is a property that describes how variables should be linked by the linker. Should a variable be available for another file to use? Should a variable be used only in the file declared? Both are decided by linkage.

Linkage thus allows you to couple names together on a per file basis, scope determines visibility of those names.

There are 2 types of linkage:

## Internal Linkage:

An identifier implementing internal linkage is not accessible outside the translation unit it is declared in. Any identifier within the unit can access an identifier having internal linkage. It is implemented by the keyword static. An internally linked identifier is stored in initialized or uninitialized segment of RAM. (note: static also has a meaning in reference to scope, but that is not discussed here).

*Animals.cpp*

// C code to illustrate Internal Linkage

#include <stdio.h>

static int animals = 8;

const int i = 5;

int call\_me(void) {

printf("%d %d", i, animals);

}

The above code implements static linkage on identifier animals. Consider Feed.cpp is located in the same translation unit.

*Feed.cpp*

// C code to illustrate Internal Linkage

#include <stdio.h>

int main() {

call\_me();

animals = 2;

printf("%d", animals);

return 0;

}

On compiling Animals.cpp first and then Feed.cpp, we get

Output : 5 8 2

Now, consider that Feed.cpp is located in a different translation unit. It will compile and run as above only if we use #include "Animals.hpp".

Consider Wash.cpp located in a 3rd translation unit.

*Wash.cpp*

// C code to illustrate Internal Linkage

#include <stdio.h>

#include "animal.cpp" // note that animal is included.

int main() {

call\_me();

printf("\n having fun washing!");

animals = 10;

printf("%d\n", animals);

return 0;

}

On compiling, we get:

Output : 5 8

having fun washing!

10

There are 3 translation units (Animals, Feed, Wash) which are using animals code.

This leads us to conclude that each translation unit accesses it’s own copy of animals. That is why we have animals = 8 for Animals.cpp, animals = 2 for Feed.cpp and animals = 10 for Wash.cpp. A file. This behavior eats up memory and decreases performance.

Another property of internal linkage is that it is only implemented when the variable has global scope, and all constants are by default internally linked.

**Usage:** As we know, an internally linked variable is passed by copy. Thus, if a header file has a function fun1() and the source code in which it is included in also has fun1() but with a different definition, then the 2 functions will not clash with each other. Thus, we commonly use internal linkage to hide translation-unit-local helper functions from the global scope. For example, we might include a header file that contains a method to read input from the user, in a file that may describe another method to read input from the user. Both of these functions are independent of each other when linked.

## External Linkage

An identifier with external linkage can be seen and used both from the file in which it is defined, and from other code files (via a forward declaration). In this sense, identifiers with external linkage are truly “global” in that they can be used anywhere in your program!

**Functions have external linkage by default**

**Global variables with external linkage**

Global variables with external linkage are sometimes called external variables.

To make a global variable external (and thus accessible by other files), we can use the extern keyword to do so:

(Non-const global variables are external by default (if used, the extern keyword will be ignored).)

int g\_x { 2 }; // non-constant globals are external by default

extern const int g\_y { 3 }; // const globals can be defined as extern, making them external

extern constexpr int g\_z { 3 }; // constexpr globals can be defined as extern, making them external (but this is useless, see the note in the next section)

int main() {

return 0;

}

**Variable forward declarations via the extern keyword**

To actually use an external global variable that has been defined in another file, you also must place a forward declaration for the global variable in any other files wishing to use the variable. For variables, creating a forward declaration is also done via the extern keyword (with no initialization value).

An identifier implementing external linkage is visible to every translation unit. Externally linked identifiers are shared between translation units and are considered to be located at the outermost level of the program. In practice, this means that you must define an identifier in a place which is visible to all, such that it has only one visible definition. It is the default linkage for globally scoped variables and functions. Thus, all instances of a particular identifier with external linkage refer to the same identifier in the program. The keyword extern implements external linkage.

When we use the keyword extern, we tell the linker to look for the definition elsewhere. Thus, the declaration of an externally linked identifier does not take up any space. Extern identifiers are generally stored in initialized/uninitialized or text segment of RAM.

It is possible to use an extern variable in a local scope. This shall further outline the differences between linkage and scope. Consider the following code:

// C code to illustrate External Linkage

#include <stdio.h>

void foo() {

int a;

extern int b; // line 1

}

void bar() {

int c;

c = b; // error

}

int main() {

foo();

bar();

}

Error: 'b' was not declared in this scope

**Explanation:** The variable b has local scope in the function foo, even though it is an extern variable. Note that compilation takes place before linking; i.e scope is a concept that can be used only during compile phase. After the program is compiled there is no such concept as “scope of variable”.

During compilation, scope of b is considered. It has local scope in foo(). When the compiler sees the extern declaration, it trusts that there is a definition of b somewhere and lets the linker handle the rest.

However, the same compiler will go through the bar() function and try to find variable b. Since b has been declared extern, it has not been given memory yet by the compiler; it does not exist yet. The compiler will let the linker find the definition of b in the translation unit, and then the linker will assign b the value specified in definition. It is only then that b will exist and be assigned memory. However, since there is no declaration given at compile time within the scope of bar(), or even in global scope, the compiler complains with the error above.

Given that it is the compiler’s job to make sure that all variables are used within their scopes, it complains when it sees b in bar(), when b has been declared in foo()‘s scope. The compiler will stop compiling and the program will not be passed to the linker.

We can fix the program by declaring b as a global variable, by moving line 1 to before foo‘s definition.

Let us look at another example

// C code to illustrate External Linkage

#include <stdio.h>

int x = 10;

int z = 5;

int main() {

extern int y; // line 2

extern int z;

printf("%d %d %d", x, y, z);

}

int y = 2;

Output: 10 2 5

We can explain the output by observing behavior of external linkage. We define 2 variables x and z in global scope. By default, both of them have external linkage. Now, when we declare y as extern, we tell the compiler that there exists a y with some definition within the same translation unit. Note that this is during the compile time phase, where the compiler trusts the extern keyword and compiles the rest of the program. The next line, extern int z has no effect on z, as z is externally linked by default when we declared it as a global variable outside the program. When we encounter printf line, the compiler sees 3 variables, all 3 having been declared before, and all 3 being used within their scopes (in the printf function). The program thus compiles successfully, even though the compiler does not know the definition of y.

The next phase is linking. The linker goes through the compiled code and finds x and z first. As they are global variables, they are externally linked by default. The linker then updates value of x and z throughout the entire translation unit as 10 and 5. If there are any references to x and z in any other file in the translation unit, they are set to 10 and 5.

Now, the linker comes to extern int y and tries to find any definition of y within the translation unit. It looks through every file in the translation unit to find definition of y. If it does not find any definition, a linker error will be thrown. In our program, we have given the definition outside main(), which has already been compiled for us. Thus, the linker finds that definition and updates y.

# References

<https://www.geeksforgeeks.org/variables-in-c/?ref=lbp>

<https://www.geeksforgeeks.org/g-fact-19-redeclaration-of-global-variable-in-c/>

<https://www.geeksforgeeks.org/variable-name-not-start-numbers-c/>

<https://www.geeksforgeeks.org/internal-linkage-external-linkage-c/>

<https://www.learncpp.com/cpp-tutorial/variable-shadowing-name-hiding/>

# END