Variables

Overview

A **variable** provides us with named storage that our programs can manipulate. Each variable has a type, the type determines the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable.

Variable Definitions

A simple variable definitions consist of a *type specifier*, followed by a list of one or more variable names separated by commas, and end with a semicolon.

```
<type_specifier> <variable_name>;
```

```
int number;
char letter;
double point;
bool true_or_false;
std::string words;
```

Initializers

An object/variable that is initialized gets the specified value at the moment it is created. The value used to initialize a variable can be arbitrary complicated expressions.

```
double price = 69.99, discount = price * 0.01;
```

> **⚠ Warning**
>
> Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replace that value with a new one.

Identifiers

Identifier composed of letters, digits, and the underscore character `_`. The language impose no limit on name length. Identifiers must begin with either a letter on an underscore. Identifiers are case sensitive; upper and lowercase letters are distinct

```
int somename, someName, SomeName, SOMENAME;
```

Convention for Variable names

- An identifier should give some indication of its meaning.
- Variable names normally are lowercase - index, not Index or INDEX.
- classes usually begin with an uppercase letter.
- Identifier with multiple words should visually distinguish each word, for example, `student_loan` or `studentLoan`, not `studentloan`.

You better avoid using this as identifiers

Table 2.3: C++ Keywords

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Scope of a Name

At any particular point in a program, each name that is in use refers to a specific entity—a variable, function, type, and so on. However, a given name can be reused to refer to different entities at different points in the program.

A **scope** is a part of the program in which a name has a particular meaning. Most scopes in C++ are delimited by curly braces.

Take a long of this program:

```cpp
#include <iostream>
int main()
{
        int sum;
        for(int val = 1; val <= 10; val++)
                sum += val;
        std::cout << "Sum of 1 to 10 is " << sum << std::endl;
        return 0;
}
```

This program defines three names: `main`, `sum`, and `val`. The `main` is accessible throughout the code, thus we can refer to it as **global scope**, this is usually the case for all functions. The `sum` is only accessible inside the `main` function, we can refer to it as **block scope**. Lastly, the `val` is defined inside the `for` statement, it can be used inside the `for` but not elsewhere in `main`.

Nested Scopes

Basically a scope inside a scope. The contained scope is referred to as an **inner scope**, the containing scope is the **outer scope**.
Take a look of this program

```cpp
#include <iostream>
int reused = 42;
int main()
{
        int unique = 0;
        //output 1
        std::cout << reused << " " << unique << std::endl;
        int reused = 0;
        // output 2
        std::cout << reused << " " << unique << std::endl;
        // output 3
        std::cout << ::reused << " " << unique << std::endl;
        return 0;
}
```

The output 1 will print `42 0` because the `reused` is defined.
The output 2 will print `0 0` because there is a local variable that will override the value of `reused`.
The output 3 will print `42 0` because it uses the scope operator to override the default scoping rules. The global scope has no name. Hence, when the scope operator has an empty left-hand side, it is a request to fetch the name on the right-hand side from the global scope. Thus, this expression uses the global `reused`.

⚠ Warning

Is is almost always a bad idea to define a local variable with the same name as a global variable that the function uses or might use.