Types and variables

Variables

A *variable* in any programming language is a named piece of computer memory, containing some information inside. Think of a variable as a box with a name, where we can "store" something. We create, edit and delete variables, as much as we need in our tasks. Variables usually are of a certain type (which define their logical representation and size). Say, we created a variable with name **A** and of type **integer**. This tells the compiler that we reserved a part of our memory for storing an integer number (usually 4 bytes long, depending on our language, compiler and machine type). We could also create another one, called **B**, but this one of type **real**. This variable will be also 4 bytes long, but we use this byte for storing floating point value, with certain precision and exponent. Variables can be static and dynamic.

Declaring Variables

Here is a list of some fundamental data types, and their characteristics, from *C++*:

Data types

Name	Description	Size	Range
char	Character and/or small integer.	1byte	-128 to 127(unsigned: 0 to 255)
int	Integer	4bytes	-2147483648 to 2147483647(unsigned:0 to 4294967295)
bool	Boolean value, can take two values "True" or "False	1bit	True or False
float	Floating point number	4bytes	3.4e +/- 38 (7 digits)
wchar_t	Wide character	2bytes	1 wide character
mixed	Any type	Unknown	Unknown

Integer types

Integers are ordinary whole numbers, or numbers with no fractional/decimal parts, and what we commonly use from day to day. Integers can be added, subtracted and multiplied. Care must be taken when they are divided, as the division of two integers is not necessarily another integer.

Integers can be signed or unsigned. The sign information takes up just one bit, but that bit reduces the absolute precision. Signed integers can represent about 1/2 of the numbers its unsigned cousin can, but can represent both positive and negative values of those numbers.

```
48
3
80398309843098340
```

Reals

Reals, or floating point numbers represent numbers that can have fractional or decimal parts. Floating point numbers are represented a bit differently than Integers, and can represent a much larger number of numbers.

Some care must be exercised with floating point numbers, because while there are an infinite number of floating point numbers, there are (by definition) only a finite number of numbers that can be expressed in a computer. Another issue is the absolute size of the numbers being handled. These issues are known as accuracy and precision. For balancing a checkbook they will not be significant, but in other cases this can be significant.

```
1.0
45.5
4.32e10
```

Characters and strings

Up until now, we've discussed types that are of a single value. Text, which is what you are reading now, isn't a single value -- it's made up of letters, each letter is an individual value. A Character is a single letter, number, punctuation or other value.

A String is a collection of these character values, often manipulated as if it were a single value.

```
characters
p
4
%
```

```
strings
Hello
blah blah
That is a nice new pair of shoes you are wearing
```

Using Variables

Visibility and Lifetime of Variables

Variables can have different visibility, or scope, according to how they are declared and used. Scope refers to the ability to access the contents of a particular variable at a certain point in the program.

Usually, a variable's scope is determined by its enclosing block. A block can be enclosed explicitly in some languages (begin/end, {/} pairs), while in others the block is implicit.

Almost all languages have an overall bounding scope, and if a program declares a variable in this scope, it is known as a **global** variable.

A **local** variable, by contrast, is one that is only visible, or "in scope", within its block.

Consider this code:

```
integer globalVariable
globalVariable = 5

begin
    integer localVariable
    localVariable = 7
    globalVariable = 8
end
```

```
 localVariable = 3  /* <-- this is a syntax error */
```

the global variable is declared outside of any block, so is in the (implicit) global scope, making it a global variable. the local variable is declared in the begin/end block, limiting its scope to that block.

Variable globalVariable starts as 5, is assigned 8 in the block and exits as 8. Variable localVariable is assigned 7, then goes out of scope before being illegally assigned 3. There is no variable localVariable at this level.

The lifetime of a variable tells you when the variable is active. In the preceding example, a global variable is created at the start of the program and has a lifetime equal to the length of the program. For localVariable, it isn't created until just after the block begins directive. Its life is over when the program passes the end directive.

Dynamic memory allocation

Many programs have very specific purposes and need very specific and small areas for memory. If we were writing a program for reading image files, we aren't sure exactly how big the images could be. Some could be hundreds of megabytes, while some might be kilobytes or less. Allocating hundreds of megabytes that aren't needed would make the program difficult to run on some computers while allocating too little memory would mean that there are some images we can't handle.

A good approach to this problem is allocating memory as needed. This is called Dynamic memory allocation. We simply ask for as much as we need while the program is running (rather than at compile time), no more and no less. In our example, this would let us edit both huge and tiny image files in an efficient and effective way. There is some added complexity since it is no longer the language that is dealing with the details of allocation, but us directly. In our example, if we are running on a lower end computer, there may not be enough memory to accommodate large image files, so we have to detect (and possibly gracefully fail) in such situations.

Dynamic memory allocation is a principal source of "memory leaks", where programs allocate, use and are then done with an area of memory, but fail to mark it as being available once again. These problems can be hard to track down since unlike other bugs, leaks aren't readily apparent.

Some modern languages have built-in mechanisms to try and prevent memory leaks, known as **garbage collection**. This system keeps track of what allocation is used where, when it is out of scope. Garbage collection has a slight performance penalty, but it's now commonly regarded as progress. Java and Python, for example, have a garbage collector.

- Next Lesson

- Previous Lesson
- Course Home Page

Retrieved from "https://en.wikiversity.org/w/index.php?title=Types_and_variables&oldid=2112895"

This page was last edited on 29 December 2019, at 02:36.

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.