Objects, values, & types

Michael Nowak

Texas A&M University

Overview

Overview

Overview

Motivation for study

- Computer memory doesn't know what type of data it stores
- The bits of memory only get meaning when we decide how that memory is to be interpreted
- This is similar to what we do everyday when we use numbers
  - What does 12.5 mean?
  - $12.5 or 12.5 cm or 12.5 gallons
  - Only when we supply the unit does 12.5 mean anything

Motivation for study

0	1	0	1	1	0	0	0

- For instance, what does the sequence of bits presented above represent?

Motivation for study

0	1	0	1	1	0	0	0

- For instance, what does the sequence of bits presented above represent?
  - As an `int`eger, the value 88

Motivation for study

0	1	0	1	1	0	0	0

- For instance, what does the sequence of bits presented above represent?
  - As an `integer`, the value 88
  - As a `character` encoded in `ASCII`, X

Motivation for study

0	1	0	1	1	0	0	0

- For instance, what does the sequence of bits presented above represent?
  - As an `int`eger, the value 88
  - As a `char`acter encoded in `ASCII`, *X*
  - As a floating-point number with an exponent range of -1 to 1 and five bits for the mantissa, 3.5

Motivation for study

0	1	0	1	1	0	0	0

- The meaning of bits in memory is completely dependent on the `type` used to access it

Overview

Basic terminology

Type Defines a set of possible values and a set of operations for an object

Basic terminology

Type Defines a set of possible values and a set of operations for an object

Object Memory that holds a value of a given type

Basic terminology

Type Defines a set of possible values and a set of operations for an object

Object Memory that holds a value of a given type

Value Set of bits in memory interpreted according to type

Basic terminology

Type
: Defines a set of possible values and a set of operations for an object

Object
: Memory that holds a value of a given type

Value
: Set of bits in memory interpreted according to type

Variable
: Named object

Basic terminology

Type	Defines a set of possible values and a set of operations for an object
Object	Memory that holds a value of a given type
Value	Set of bits in memory interpreted according to type
Variable	Named object
Declaration	Statement that gives a name to an object

Basic terminology

Type	Defines a set of possible values and a set of operations for an object
Object	Memory that holds a value of a given type
Value	Set of bits in memory interpreted according to type
Variable	Named object
Declaration	Statement that gives a name to an object
Definition	Declaration that sets aside memory for an object

Overview

Thinking about `objects`, `types`, and `values`

- Informally, we can think of an `object` as a box
- Into which we can put `value`s of a given `type`
- An `int` box can hold integers, such as 7, 42, and -399
- A `std::string` box can hold character string values, such as "Computer Science", "Texas A&M University", and "Gig 'em"

Thinking about `objects`, `types`, and `values`

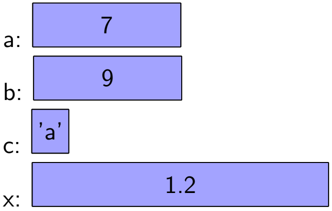
- Graphically, we can informally think of it like this:

```
int a = 7;

int b = 9;

char c = 'a';

double x = 1.2;
```



- Note: different `types` of `objects` take up different amounts of space
  - The compiler sets aside the same fixed amount of storage for each `object` of a specified primitive built-in `type`

Overview

Primitive built-in types

- The primitive built-in `types` are the most basic elements from which our C++ programs are constructed from; included are:
  - A Boolean type (i.e., `bool`)
  - Character types (e.g., `char`)
  - Integer types (e.g., `int`)
  - Floating-point types (e.g., `double`)
- The Boolean, character, and integer types are known as the `integral types`
- Together, the `integral types` and `floating-point types` are known as the `arithmetic types`

Primitive built-in types

- As we will see, the integral and floating-point `types` come in different flavors to give the user a choice in:
  - the amount of storage consumed
  - the range available for `values`
  - and precision
- In this course, the following types will *usually* be sufficient:
  - `bool` for logical values
  - `char` for characters
  - `int` for integer values
  - `double` for floating-point values

Primitive built-in types

- As we will discuss later, other types can be constructed from the primitive built-in `type`s, including:
  - Pointer types (e.g., `int*`)
  - Array types (e.g., `char[]`)
  - Reference types (e.g., `int&`)
  - Data structures and classes

Overview

Boolean (bool) type

- The possible values of a Boolean (i.e., `bool`) type are `true` and `false`
- This type is primarily used to express the result of logical operations

```
bool res = x == y; // = is assignment; == is equality
```

- In both arithmetic and logical expressions,
  - `bool`s are converted to integers
  - arithmetic and/or logical operations are performed on the converted values
  - If the result is converted back to `bool`, a nonzero value is converted to `true` whereas a zero value to `false`

```
bool x = true;
bool y = true;
bool z = x + y;
cout << (x + x + y + y);
```

Boolean (bool) type

- By definition, true has the value 1 when implicitly converted to an integer; false has the value 0

```
int i = true; // int(true) is 1; i is initialized to 1
```

- Integers can be implicitly converted to bool values: nonzero integers convert to true; 0 converts to false

```
bool b = 11; // bool(11) evaluates true; b is initialized to true
```

Overview

Character (`char`) types

- The `char` type can hold a character of the implementation's character set

```
char ch = 'c';
```

- Each character constant has an integer value; however, whether `char` is signed or unsigned is implementation-defined
  - `signed char` can hold at least the values -127 to 127
  - `unsigned char` can hold at least 0 to 255
- Are integral types, so arithmetic and logical operations apply

Character (`char`) types

- Safe to assume the implementation character set includes:
  - 26 alphabetic characters of English
  - Decimal digits (0-9)
  - Basic punctuation characters
- It is not safe to assume that there are:
  - No more than 127-characters in an 8-bit character set
  - No more alphabetical characters than that provided by English language
  - That the alphabetical characters are contiguous
    - EBCDIC has a gap between 'i' and 'j'
  - That every character used to write C++ is available

Character literals

- A literal is a notation for representing a fixed value; `character literals` are also known as `character constants`
- A character literal is a character enclosed by single quotes
  - 'c'
  - '9'
  - '.'
- Are really symbolic constants for the integer value of the respective character in the implementation's character set
- Some characters have names that use backslash as an escape character

Overview

Integer Types

- There are three integer types that vary from one another in size:
  - `short` int
  - "plain" `int`
  - `long` int
- Each integer type comes in three forms:
  - "plain" `int`
  - `signed int`
  - `unsigned int`
- Usually, it is not a good idea to use an `unsigned int` instead of an `int` to gain one more bit to represent positive numbers
- Regardless of implementation, "plain" `int`s are always signed

Integer literals

- Are available to us in four forms:
  - Decimal
  - Octal
  - Hexadecimal
  - Character Literals
- A literal prefixed with `0x` is a hexadecimal (base-16) number
- A literal starting with a `0` (and not proceeded by an x) is an octal (base-8) number
- The suffix `U` can be used to write `unsigned` literals
- The suffix `L` can be used to write `long` literals
- If no suffix is applied, the compiler will produce an integer literal of suitable type based on value and size of the implementation's integer types

Overview

Floating-point types

- Represent floating-point numbers
- There too are three floating-point types that vary from one another in size:
  - `float` (single-precision)
  - `double` (double-precision)
  - `long double` (extended-precision)
- The exact meaning of `single`-, `double`-, and `extended`-precision are implementation defined

Floating-point literals

- The default floating-point literal type is `double`
- If you'd like a `float` floating-point literal, you must suffix the literal with `F`
- Similarly, if you'd like a `long double` floating-point literal, you must suffix the literal with `L`

Overview

Variables

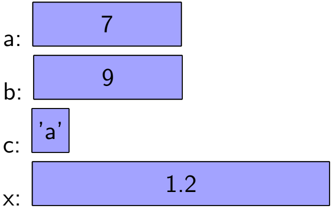
- A program `variable` is an `abstraction` of a computer memory cell or collection of program memory cells

```
int a = 7;

int b = 9;

char c = 'a';

double x = 1.2;
```



`Variables`

- Programmers often think of `variables` as `names` for memory locations, but there is much more to a `variable` than just a `name`
- A `variable` can be characterized as a sextuple of attributes:
  - `Name`
  - `Address`
  - `Value`
  - `Type`
  - `Lifetime`
  - `Scope`

Overview

Names

- A `variable`'s `name` is composed of a sequence of letters and digits
  - The first character of an identifier must be a letter
  - Uppercase and lowercase letters are distinct; C++ identifiers are case-sensitive
  - Underscore character "_" is considered a letter; however, names started with an underscore are reserved for facilities in the implementation
  - While C++ does not impose a limit on the number of characters in an identifier, some parts of an implementation not under control of the compiler sometimes do
  - Some implementations are more restrictive in the characters accepted in an identifier
  - C++ "keywords" cannot be used for our names; a list of these words are provided on page A.3.1 of your Stroustrup text

Overview

Address

- The address of a variable is the machine memory address with which it is associated
- Sometimes called a variable's l-value, because the address is what is required when the name of a variable appears on the left side of assignment
- It is possible to have multiple namess associated with the same address
  - When more than one name can be used to access the same memory location, such names are called aliases
  - If total and sum are aliases, any change to the value of total also changes the value of sum and vice versa

Overview

`Type`

- The `type` of a variable determines the
  - range of values the variable can store, and
  - the set of operations that are defined for the values of that type

Overview

Value

- The `value` of a variable is the contents of the memory cell or cells associated with the variable
- Sometimes called a `variable`'s `r-value` because it is what is required when the name of the variable appears in the right side of an assignment statement
  - To access the `r-value`, the `l-value` must be determined first; such determinations are not always trivial

Overview

Lifetime

- A `binding` is an association between an attribute and an entity, such as between a variable and its type or value, or between an operation and a symbol
- The memory cell to which a `variable` is `bound` is taken from a pool of available memory
  - This process is called `allocation`
  - `Deallocation` is the process of placing a memory cell that has been unbound from a variable back into the pool of available memory
- The `lifetime` of a variable is the time during which the variable is bound to a specific memory location
  - Begins when the `variable` is `bound` to a specific cell
  - Ends when the `variable` is `unbound` from that cell

Overview

Scope

- A `scope` is a part of the program in which a `name` has a particular meaning
  - In C++, most `scopes` are delimited by curly braces
- The same `name` can refer to different entities in different `scopes`
- `Names` are `visible` from the point where they are `declared` until the end of the `scope` in which their declaration appears
- A `name` is `visible` in a `statement` if it can be referenced or `assigned` in that statement
- A `variable` is `local` in a program unit or block if it is `declared` there
- A `variable` is `non-local` in a program unit of block if it is `visible` within that region of the program but is not `declared` there

Scope

- So, once we provide a `name` to an `object`, that `name` is restricted to the part of the program in which it is `declared`
- In other words, a declaration introduces a name into a `scope`

```
int x = 10; // global variable
int main() {
    x += 1; // OKAY: x = x + 1 = 11
    {
        int y = x; // use global x to initialize; y = 11
        int x = 2; // local variable x initalized to 2; global x is hidden
        y += x; // OKAY: y is assigned the value of y + local x = 11 + 2 = 13
        y += ::x; // OKAY: y is assigned value of y + global x = 13 + 11 = 24
    }
    y += 1; // ERROR: y is not declared in this scope
}
```

Overview

Declarations

- `Names` are a lot easier to remember than `addresses`; therefore, we frequently use `variables` to access `objects` in memory
- Each `named object` (i.e., a `variable`) has a specific `type` associated with it, which determines the `values` that be put into it
- Without the specification of a `type`, we would be dealing with only bits of memory; the `type` denotes how those bits are to be interpreted

Declarations

- Before a `name` can be used (including `variable` identifiers), we must inform the `compiler` of its `type` through a `declaration`
- Most `declarations` are also `definitions`, which define the entity for which the `name` will refer (cause memory to be allocated)
  - This is the case for the built-in `arithmetic types`
- There must always be exactly one `definition` for each named entity in our programs; however, we can have multiple `declarations` (but each must agree on the type of the `identifier`)

Overview

Declaration structure

- A declaration is comprised of four parts:
  - An optional specifier
    - An initial keyword that specifies some non-type attribute
    - E.g., `virtual` or `extern`
  - A base type
  - A declarator
    - Composed of a name and optionally some declarator operators that are either prefix or postfix; most common declarator operators include:

*	pointer	prefix
*const	constant pointer	prefix
&	reference	prefix
[]	array	postfix
()	function	postfix

    - Postfix declarator operators bind more tightly than prefix ones
    - Declarator operators apply to individual names only

```
int x, y // int x; int y
int* x, y; // int* x, int y; NOT int* y
int x, *q; // int x, int* y;
```

  - An optional initializer

Overview

Initialization

- Initialization (”starts out with”): giving a variable its initial value; has type specification
- When an initializer is specified in the declaration, the initializer determines the initial value of an object

```
int x; // x is initialized to 0
int main() {
    int y; // y does not have a well-defined value
    return 0;
}
```

  - When no initializer is specified for a global, namespace, or local static object, initialization will be the type’s zero value
  - When no initializer is present for local variables (and objects created on the free store), the variable will not contain a well-defined value

Overview

Assignment

- Assignment ("gets"): giving a variable a new value; does not have type specification

```
int main() {
    int z = 10; // z starts out with 10; initialization
    z = 12; // z gets the value 12; assignment
    return 0;
}
```

Overview

References

- Lippman, B., Lajoie, Josee, & Moo, B. E. (2016). *C++ primer* (5th ed.). Addison-Wesley.
- Sebesta, R. W. (2016). *Concepts of programming languages* (11th ed.). Pearson Education.
- Stroustrup, B. (2014). *Programming: principles and practice using C++* (2nd ed.). Addison-Wesley.