Things you're already familiar with, but now in C++ with more detail

Michael Nowak

Texas A&M University

Overview

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions

Statements

Function basics

Simple input and output

References

Overview

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions

Statements

Function basics

Simple input and output

References

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

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions

Statements

Function basics

Simple input and output

References

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 `object`s, `type`s, and `value`s

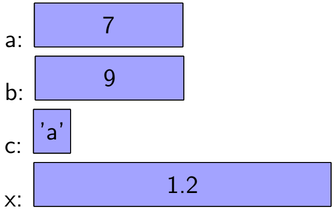
- Graphically, we can informally think of it like this:

```
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

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions

Statements

Function basics

Simple input and output

References

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

- The integral and floating-point `type`s come in different flavors to give the user a choice in:
  - the amount of storage consumed
  - the range available for `value`s
  - 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

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables
- Names
- Address
- Type
- Value
- Lifetime
- Scope

Declarations

Assignment

Expressions

Statements

Function basics

Simple input and output

References

Variables

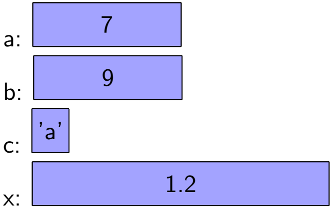
- A program variable is an abstraction of a computer memory cell or collection of program memory cells

```
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`

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
  - C++ "keywords" cannot be used for our names

`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

`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

`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

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

Scope

- A `scope` is a part of the program in which a `name` has a particular meaning
  - In C++, most `scope`s are delimited by curly braces
- The same `name` can refer to different entities in different `scope`s
- `Name`s are `visible` from the point where they are `declared` until the end of the `scope` in which their declaration appears

Scope

- 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

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations
- Declaration structure
- Initialization

Assignment

Expressions

Statements

Function basics

Simple input and output

References

Declarations

- `Name`s are a lot easier to remember than `address`es; therefore, we frequently use `variable`s to access `objects` in memory
- Each `named` `object` (i.e., a `variable`) has a specific `type` associated with it, which determines the `value`s 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`

Declaration structure

- A declaration is comprised of four parts:
  - An optional specifier
    - An initial keyword that specifies some non-type attribute
    - E.x., `const`
  - 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

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 present for local variables, the variable will not contain a well-defined value
  - When no initializer is specified for a global variable, initialization will be the type's zero value

Overview

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions

Statements

Function basics

Simple input and output

References

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

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions
- Composition of expressions
- Types of operators
- Grouping operators and operands
- Operators

Statements

Function basics

Simple input and output

References

Composition of expressions

- The smallest piece of a programming language that has meaning is called a `token`
- Many tokens in C++ are words; others are symbols like punctuation
- An `expression` is a group of `token`s that yield a result when evaluated

- In C++, some `tokens` are interpreted as `operands` in an `expression`
- The simplest form of an `expression` is composed using one or more `operands` that yield a result when evaluated

- Other `tokens` comprise `operators`
- More complicated expressions are formed by incorporating an `operator` and one or more `operands`

Composition of expressions

- The smallest piece of a programming language that has meaning is called a `token`
- Many tokens in C++ are words; others are symbols like punctuation
- An `expression` is a group of `tokens` that yield a result when evaluated

- In C++, some `tokens` are interpreted as `operands` in an `expression`
- The simplest form of an `expression` is composed using one or more `operands` that yield a result when evaluated

- Other `tokens` comprise `operators`
- More complicated expressions are formed by incorporating an `operator` and one or more `operands`

Composition of expressions

- The smallest piece of a programming language that has meaning is called a `token`
- Many tokens in C++ are words; others are symbols like punctuation
- An `expression` is a group of `tokens` that yield a result when evaluated

- In C++, some `tokens` are interpreted as `operands` in an `expression`
- The simplest form of an `expression` is composed using one or more `operands` that yield a result when evaluated

- Other `tokens` comprise `operators`
- More complicated expressions are formed by incorporating an `operator` and one or more `operands`

Types of operators

- `Unary operators` act on one `operand`
- `Binary operators` act on two `operands`
- Some tokens are used as both `unary operators` and `binary operators`

Types of operators

- `Unary operators` act on one `operand`
- `Binary operators` act on two `operands`
- Some tokens are used as both `unary operators` and `binary operators`
- There is even a `ternary operator` in C++; more on that later

Grouping operators and operands

- An expression with two or more operators is a `compound expression`
- Understanding the evaluation of `compound expressions` requires an understanding of
  - `precedence`
  - `associativity`
  - `order of evaluation`

Precedence

- Operands of operators with higher precedence group more tightly than those at lower precedence
  - Multiplication and division both have higher precedence than addition and subtraction
  - Multiplication and division group before operands to addition and subtraction

$$3 + 4 * 5 = 23 \text{ not } 35$$

Associativity

- Associativity determines how operators of the same precedence are grouped
  - Assignment operators are right associative, which means operators at the same precedence group right to left

```
int ival, jval;
ival = jval = 0;
```

  - Arithmetic operators are left associative, which means operators at the same precedence group left to right

$$20 - 15 - 3 = 2 \text{ not } 8$$

Order of evaluation

- Precedence specifies how the operands are grouped
- Precedence does not specify the order in which the operands are evaluated
- In most cases, the order is largely unspecified
- For example,

$$int\ i = f1() + f2() * f3();$$

- `f2` and `f3` must be called before multiplication can be done
- However, it is unknown whether `f1` will be called before `f2` or vice versa
- We then add the result of f1() to the product of `f2` and `f3`

Arithmetic operators (Left Associative)

Operator	Function	Use
`+`	unary plus	`+ expr`
`-`	unary minus	`+ expr`
`*`	multiplication	`expr * expr`
`/`	division	`expr / expr`
`%`	remainder	`expr % expr`
`+`	addition	`expr + expr`
`-`	subtraction	`expr - expr`

Logical and relational operators

<table>
<tr><th>Associativity</th><th>Operator</th><th>Function</th><th>Use</th></tr>
<tr><td>Right</td><td>!</td><td>logical NOT</td><td>!expr</td></tr>
<tr><td>Left</td><td><</td><td>less than</td><td>expr < expr</td></tr>
<tr><td>Left</td><td><=</td><td>less than or equal</td><td>expr <= expr</td></tr>
<tr><td>Left</td><td>></td><td>greater than</td><td>expr > expr</td></tr>
<tr><td>Left</td><td>>=</td><td>greater than or equal</td><td>expr >= expr</td></tr>
<tr><td>Left</td><td>==</td><td>equality</td><td>expr == expr</td></tr>
<tr><td>Left</td><td>!=</td><td>inequality</td><td>expr != expr</td></tr>
<tr><td>Left</td><td>&&</td><td>logical and</td><td>expr && expr</td></tr>
<tr><td>Left</td><td>||</td><td>logical or</td><td>expr || expr</td></tr>
</table>

Overview

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions

Statements
- Types of statements
- Conditional statements
- Iterative statements

Function basics

Simple input and output

References

Simple statements

- Most `statements` in C++ end with a `semicolon`
- A `statements` becomes an `expression statement` when it is followed by a `semicolon`

$$3 + 5;$$

$$std :: cout << (2 + 3);$$

Null statements

- The simplest `statement` is the `null statement`
- Useful when the language requires a statement, but your logic does not

```
;
```

Compound statements

- A `compound statement` is usually referred to as a `block`
- It is a (possible empty) sequence of statements and declarations surrounded by a pair of curly braces
- Used when the language requires a single statement, but the logic of our program requires more than one
- `Compound statements` are *not* terminated by a `semicolon`

Conditional statements

- C++ provides two statements that allow for conditional execution
  - The `if statement`
  - The `switch statement`

The `if` statement

- An `if statement` conditionally executes another statement based on whether a specified condition is true
- Two forms:
  - `Syntactic` form of the simple `if` is

    ```
    if (condition)
        statement
    ```

  - An `if else statement` has the form

    ```
    if (condition)
        statement
    else
        statement
    ```

Iterative statements

- Provide for repeated execution until a condition is true

`while` statement

- Repeatedly executes a statement as long as a condition is true
- Syntactic form is

```
while (condition)
    statement
```

- In a `while`, the statement (which is often a `block`) is executed as long as `condition` evaluates to `true`
- Usually, the `condition` or the `loop body` must do something to change the value of the expression

`while` statement

- Frequently used when we want to iterate indefinitely, for example
  - While reading input
  - When we need to access the value of the loop `control variable` outside of the loop.

`for` statement

- Syntactic form is

```
for (init-expression; condition; expression)
    statement
```

- The `for` and part inside the parentheses is often referred to as the `for` header
- `init-expression` must be either a declaration statement, an expression statement, or a null statement (each of which end with a `semicolon`)
- The statement (which is often a `block`) is executed as long as `condition` evaluates to `true`
- `expression` is evaluated after each iteration of the loop

`for` statement

- Provided the following `for` loop,
```
for (int i = 0; i != 10; ++i)
    std::cout << i << std::endl;
```
  1. `init-expression` is executed once at the start of the loop
  2. Next, the `condition` is evaluated.
     - If it is true, the `loop body` is executed
     - otherwise, the loop terminates
  3. If the `condition` was true, the `statement` is executed
  4. The `expression` is evaluated and we continue from step 2

`do while` statement

- Syntactic form is

```
do
    statement
while(condition);
```

- The `do while` `statement` is like a `while` `statement`, but has its `condition` tested after the `statement` completes
- Regardless of the value of the condition, the `loop body` is executed at least once
- If `condition` evaluates to false, then the loop terminates; otherwise, the loop is repeated

Overview

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions

Statements

Function basics

Simple input and output

References

Function basics

- A `function declaration` introduces a function's name to the compiler, and tells its return type and parameter list; syntax follows scheme of regular declaration
  - base type is the return type; if function does not return a value specify type `void`
  - declarator is composed of an identifier (name of function) and a postfix declarator operator, the parameter list (`()`)
  - terminated with a semi-colon
  - together, the identifier and parameter list are called the `function signature`
  - the function signature and return type is known as the `function header`
- Function definition provides the actual implementation of the function; syntax is function header – as it is written in the declaration – followed by what's known as the function body
  - the function body is composed of a sequence of statements that together define that function's behavior.

Simple function example

```cpp
#include <iostream>

// Function declaration for max
int max(int, int);

int main()
{
    int maxValue = max(11, 7); // invokes function
        max with arguments 11 and 7 that initialize
        parameters a and b respectively.
    return 0;
}

// Function definition for max
int max (int a, int b)
{
    if (a < b)
        return b;
    else
        return a;
}
```

Overview

- Basic terminology
- Thinking about objects, types, and values
- Primitive built-in types
- Variables
- Declarations
- Assignment
- Expressions
- Statements
- Function basics
- Simple input and output
  - Reading from standard input with std::cin
  - Writing to standard output with std::cout
- References

Reading from standard input with std::cin

- We can read keyboard input from the terminal window through `std::cin`
- `std::cin` is used with the extraction operator (`>>`) along with the name of the variable to which we'd like to store the data read
  - `int i = 0; double d = 0.0;`
    `std::cin >> i >> d;`
    - Reads an integer followed by a floating-point value (need whitespace between the two values)
    - ex. `11 3.14`
  - The input must match the type of the variable where the data is to be stored (ex.,type of `i` above)
  - `std::cin` is whitespace deliminted (whitespaces, tabs, new-line...); whitespace characters terminate the value being extracted

Reading from standard input with std::cin cont.

- Suppose we enter `3*4+8` to standard input

Reading from standard input with std::cin cont.

- Suppose we enter `3*4+8` to standard input
- This would be represented as a stream of characters as the data flowed from the keyboard to our program

Reading from standard input with std::cin cont.

- Suppose we enter `3*4+8` to standard input
- This would be represented as a stream of characters as the data flowed from the keyboard to our program
- We would specify how we would like to consume these five characters using `std::cin` in our program

Reading from standard input with std::cin cont.

- Suppose we enter `3*4+8` to standard input
- This would be represented as a stream of characters as the data flowed from the keyboard to our program
- We would specify how we would like to consume these five characters using `std::cin` in our program
  - We could read the integer `3`, followed by the character `*`, etc.

Reading from standard input with std::cin cont.

- Suppose we enter `3*4+8` to standard input
- This would be represented as a stream of characters as the data flowed from the keyboard to our program
- We would specify how we would like to consume these five characters using `std::cin` in our program
  - We could read the integer `3`, followed by the character `*`, etc.
  - Perhaps we could read the whole sequence of characters (`3*4+8`) at oncem, given that there are no whitespaces between them

Reading from standard input with std::cin cont.

- Suppose we enter `3*4+8` to standard input
- This would be represented as a stream of characters as the data flowed from the keyboard to our program
- We would specify how we would like to consume these five characters using `std::cin` in our program
  - We could read the integer `3`, followed by the character `*`, etc.
  - Perhaps we could read the whole sequence of characters (`3*4+8`) at oncem, given that there are no whitespaces between them
    - We'll cover how to do this later

Reading from standard input with std::cin cont.

- Suppose we enter `3*4+8` to standard input
- This would be represented as a stream of characters as the data flowed from the keyboard to our program
- We would specify how we would like to consume these five characters using `std::cin` in our program
  - We could read the integer `3`, followed by the character `*`, etc.
  - Perhaps we could read the whole sequence of characters (`3*4+8`) at oncem, given that there are no whitespaces between them
    - We'll cover how to do this later
- It is completely up to us what type we would like to convert the characters into

  (as long as the character sequence is valid for the desired type)

Writing to standard output with std::cout

- We can write data to the terminal window through `std::cout`
- `std::cout` is used with the insertion operator along with the name of the variable or literal values that we'd like to write
  - ```
    int i = 11;
    std::cout << i << " Hello, World! " << 3.14;
    ```
  - This writes `11 Hello, World! 3.13` to standard output.
    ```

Overview

Basic terminology

Thinking about objects, types, and values

Primitive built-in types

Variables

Declarations

Assignment

Expressions

Statements

Function basics

Simple input and output

References

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.