IM1003: Programming Design

Basic Data Types and Operations

Ling-Chieh Kung

Department of Information Management
National Taiwan University

February 25, 2013

Outline

- **Basic data types**
- Operations, expressions, and statements
- Operators
- Casting
- The **`cin`** object

Data type

- For our programs to do more things, there must be **variables**.
  - To do things like **`a = b + c`** and so on.
  - The way for us to access memory.
- In C++, each variable must be have its **data type**.
  - The data type determines the operations that can be done on it.
  - The data type tells the computer how to allocate memory spaces.
- Here we introduce **basic** (or **built-in** or **primitive**) data types.
  - Those provided as part of the C++ standard.
- We will discuss how to define new data types later in this semester.

Literals and variables

- Before we start, let's know distinguish **literals** from variables.
- Literals: items whose contents are **fixed**.
  - For example, 3, 8.5, and "Hello world".
  - Can be numbers, strings, and Boolean values.
- Variables: items whose values may **change**.
  - These self-defined elements must be given names.
  - Defining a new variable requires the programmer to specify its data type.

Basic data types

- There are ten basic data types, belonging to two categories.

Category	Type	Bytes	Type	Bytes
Integers	bool	1	long	4
	char	1	unsigned int	4
	int	4	unsigned short	2
	short	2	unsigned long	4
Fractional numbers	float	4	double	8

- The number of bytes is **compiler-dependent**. The values shown here are for Dev-C++ 5.4.

Variable declaration

- When we want to use a variable, we must first **declare** it.
  - We need to specify its **name** and **data type**.
- The compiler, rather than a programmer, decides how many bytes are allocated to a type.
- The statement for variable declaration is

```
type variable name;
```

  - For example, `int myInteger;` declares an integer variable called `myInteger`.

Variable declaration

- After declaration, compiler will "allocate" a space in the memory for the variable.
- A variable name is the name of that memory space.
  - We do not need to memorize the memory address (which is a sequence of numbers).
  - We may access the space through the variable name.

Declaration and assignment

- Besides declaring a variable, we may also assign values to a variable.
  - `int a;` // declare an integer variable
  - `char b;` // declare a character variable
  - `a = 10;` // **assign** `10` to `a`
  - `b = 'x';` // assign `'x'` to `b`
- We may even do these together. The assignment is then called **initialization** if done with declaration.

```
type variable name = initial value;
```

  - `int c = 0.5;` // declaration and assignment

Declaration and assignment

- Following an assignment operation, the compiler will put the value into the space for the variable.
- Without initialization, the variable may be of any value!
  - Declaration only allocate a memory space. It does not know what to do to that space.
  - Those are the values left since the last time this space is used.

Multi-variable declaration

- We may declare multiple variable in the same type together.
- For example, the syntax for declaring three variables is

```
type name 1, name 2, name 3;
```

  - There is no limit on the number of variables.
- We may initialize all of them also in a single statement:

```
type name 1 = value 1, name 2 = value 1;
```

- Personally I do not like this style, especially when we do initialization at declaration.

Rules of choosing a variable name

- Only letters, numbers, and the underline symbol (_) are allowed.
- Cannot starts with a number.
- Cannot contain a white space.
- Cannot be the same with any C++ **keywords**.
  - Usually those words colored by your editor.
  - E,g., `int`, `float`, `double`, `return`, `using`, and `namespace`.
  - Listed in Table 1.1 of the textbook.
- **Case-sensitive**.
  - In fact, the whole C++ world is case-sensitive.

Good programming style

- **Always** initialize your variables.
  - If no value fits, initialize a variable as 0.
- Use **mnemonic** (pronounced as "knee-monic") instead of a short/meaningless name.
  - `int yardToInch = 12;` is better than `int y = 12;`
- **Capitalize** the first character of each word, but not the first word.
  - `int yardToInch = 12;`
  - `double avgGrade = 0;`
  - `int maxCapacity = 100;`
- This is the so-called "camel case".

Constants

- Sometimes we want to use a variable to store a particular value.
  - In a program doing calculations regarding circles, the value of π may be used **repeatedly**.
  - We do not want to write many `3.14` throughout the program! Why?
  - We may declare `pi = 3.14` once and then use `pi` repeatedly.
- In this case, this variable is actually a **symbolic constant**.
  - We want to prevent it from being **modified**.

Constants

- A **constant** is one kind of variables.
- To declare a constant, use the key word `const`:
  - `const int a = 100;` // Declaring a constant integer
  - All further assignment operations on a constant generate compilation errors.
- You must **initialize** a constant.
  - Otherwise there is no way to assign a value to it.
- You are suggested to use **capital characters** and **underlines** to name constants. This is to distinguish them from usual variables.
  - `const double PI_CONST = 3.1416;`
  - `const int MAX_LEVEL = 5;`
  - Some people use lowercase characters and underlines.

`int`

- `int` means an integer.
- Use 4 bytes to store from -2^{31} to 2^{31}.
  - Try the example "**02_01_intLimit**".
  - On p. 39 of the textbook: `<limits.h>` or `<climits>`, **not** `<limits>`.
- `unsigned int` (4 bytes): from 0 to 2^{32}.
- `short` (2 bytes): from –32768 to 32767.
- `long`: the same as `int` in Dev-C++.
  - Try the example "**02_02_sizeof**".
- The C++ standard requires the following:
  - The space for a `long` variable is no smaller than that for an `int` one.
  - The space for an `int` variable is no smaller than that for a `short` one.

`int`

- As an example, the following codes contains two `int` variables:

```
int a = 10;
int b = 20;
cout << a + b; // result: 30
```

- Rules for writing an integer in C++:
  - Can contain only numbers, `+`, and `–`.
  - Can be `0`, `–10`, `+36`, `253`, etc.
  - Cannot be `$255`, `6.9`, `2,532`, etc.
- Be aware of **overflow**.
- `short` and `long` just create integers with different "lengths".
  - In most information systems this is not an issue.

char

- Means a character.
  - Use **one byte** (0 to 255) to store English characters, numbers, and symbols.
  - Cannot store, e.g, Chinese characters.
- It is also an "**integer**"!
  - These characters are encoded with the ASCII code in most PCs.
  - ASCII = American Standard Code for Information Interchange.
  - See the ASCII code mapping in your textbook.
  - Some encoding:

Character	A	B	Z	a	b	z	0	1	9
Code	65	66	90	97	98	122	48	49	57

  - Try the example "**02_03_char**".

Literals in `char` type

- Use single quotation marks to mark your `char` literal.
  - `char c = 'c';`
  - `char c = 99;` // 99 is c's ASCII code
- Some wrong ways of marking a character:
  - Wrong: `char c = "c";`
  - Wrong: `char c = 'cc';`

`float` and `double`

- `float` and `double` are used to declare fractional numbers.
  - Can be `+10.625`, `5.0`, `-6.2`, etc.
  - Can be `1.625e2`, `16.25e2`, `7.33e-3`, `3571.62e20`, etc.
  - Cannot be `$75.2`, `2,345.0`, `3.45.89`, etc.
- They follow the IEEE floating point standards.
  - `float` uses 4 bytes to record values between $1.4 * 10^{-45}$ and $3.4 * 10^{38}$.
  - `double` uses 8 bytes to record values between $4.9 * 10^{-324}$ and $1.8 * 10^{308}$.

Literal of `float` type

- When we write a usual fractional literal in C++, such as `3.46`, it is set to be a `double` literal (occupying 8 bytes).
- We may append `F` to make it a `float` literal.
  - `float f1 = 0.5678F;`
- double (8 bytes) is more precise than float (4 bytes).

```
double d1 = 0.5678;
double d2 = -6.789E64;
float f1 = 0.5678F;
float f2 = -6.789E64F; // error
```

- Dev-C++ (and some other compilers) offers `long double` as a 16 bytes floating point data type.

bool

- A bool variable uses 1 byte to record true or false.
  - All non-zero values are treated as true. Try the example "**02_04_bool**".
  - 7 bits are wasted.
- Will be particularly important starting from the next week.

Outline

- Basic data types
- **Operations, expressions, and statements**
- Operators
- Casting
- The `cin` object

Operations

- In C++, we may use **operations** to create more interesting things.
  - An operation combines **operators** and **operands** to generate a result.
  - It does one thing and then **return a value**.
- `cout << "Hello."` is an operation.
  - It print out `"Hello."` on the screen.
  - The operator is `<<` and the operands are `cout` and `"Hello."`.
- `cout << "Hello." << "\n";` contains **two** operations.
  - The second `<<` first concatenates `"Hello."` and `"\n"`.
  - The first `<<` then sends the string into `cout`.
- All the above operations return `void`, which means "**nothing**".

The assignment operation

- Another type of operations is the **assignment** operations.
  - An assignment operation does the assignment.
  - It then returns **the value it assigns**.
- `a = 10`: One operation which **returns 10**.
- `b = a = 10`: Two operations.
  - `b = a = 10`: The first operation is `a = 10`. This operation assigns 10 to `a`, and then return 10.
  - `b = 10`: Thus `b` is assigned 10, too.
- In fact we can do `cout << a = 10;`.
  - However, never do this. All assignment should be made as clear as possible.

Arithmetic operations and expressions

- There are also **arithmetic operations**:
  - Just like simple calculations.
  - It combines several arithmetic operators and operands.
  - It calculates the result and then returns the result.
- Some examples:
  - `3 * 5`: returns `15`.
  - `9 + 6`: returns `15`.
  - `9 + 3 * 5`: returns `24`. How many operations do we have here?
- An **expression** is a sequence of related arithmetic operations.
  - `9 + 6` is an expression. `9 + 3 * 5` is also an expression.

Statements

- One **statement** contains one or many operations.
- It will execute those operations, and then throw the value away.
- A statement ends with a semicolon (`;`) .
- Some examples:
  - `cout << 6;`
  - `cout << 6 + 9;`
  - `cout << a + b * 5;`
  - `b = c * d + f;`
- All the returned values are dropped.

Statements

- `c = a + b;`

  is the same as (but certainly better than)

```
c
    =
            a
+                       b;
```

- Only the semicolon matters.

Good programming style

- `c = a + b;` is the same as (but certainly better than)

```
c
    =
          a
+                         b;
```

  - Only the semicolon matters.
- It is recommended to use spaces in your statements:

  `c = a + b;`

  is the same as (but better than)

  `c=a+b;`.

Outline

- Basic data types
- Operations, expressions, and statements
- **Operators**
- Casting
- The **`cin`** object

Operations

- Let's introduce some most common operators.
- Recall that "operation = operand + operator".
  - The operators do something on or manipulate the operands.
- **`3 = 2 + 1`**:
  - Operand: **`3`**, **`2`**, **`1`**.
  - Operator: **`=`**, **`+`**.
  - Note: The above expression is not valid in C++. It is just used to illustrate the idea of operators and operands.

Associativity of operators

- For each operator, we specify the following:
  - The **number** of operands it operates on.
  - The **types** of operands it operates on.
  - The associativity: **Where** should it stand among those operands and the **order** it takes to operate on the operands.
- It is important to define and follow these rules carefully (though most of them are very intuitive).
  - These rules are part of the grammar of programming languages.
  - Because computers cannot read human language.

The assignment operator

- The assignment operator **`=`** assigns a value to a variable.

```
variable = expression
```

  - We call it "**assign**" or "**becomes**".
- It has nothing to do with the "equals" in mathematics.
  - "equals" will be introduced in the next week.
- For **`a = b + c`**:
  - We assign **`b`** plus **`c`** to **`a`**.
  - **`a`** becomes **`b`** plus **`c`**.
- Thus we know what does **`a = a + 1`** mean.
  - It just means that "**`a`** becomes **`a`** plus one."
- An assignment operator returns the assigned value.

The assignment operator

- The assignment operator is a **binary** operator.
  - It is associated with two operands.
- It accepts all basic data types.
  - Conversion between different data types may occur.
  - This is called "casting" and will be discussed at the end of this lecture.
- Its associativity is from right to left.
  - It **only** assigns the value at its right to the variable at its left.

Some unary operators

- **Unary** operators operates on **one** operand.
- The following unary operators are for **non-Boolean** types:
  - `+`: positive: `+3`, `+5.678` (default; do not need to use it).
  - `-`: negative: `-5`, `-5.678`.
- The following is for the **Boolean** type:
  - `!`: not: `!true` is the same as `false`.
- The must be put at the **left** of the operand.
- Each of these unary operator returns the resulting value.

Arithmetic operators

- We have all the basic arithmetic operators:
  - `+`: addition.
  - `-`: subtraction.
  - `*`: multiplication.
  - `/`: division.
  - `%`: remainder (it is called the modulus operator).
- The modulus operator finds the remainder of a division.
  - `10 % 3` results in `1`, `20 % 3` results in `2`, etc.
- They are all binary and associate operands from left to right.
  - `3 + 8 - 9` => `11 - 9` => `2`.
- Each of these arithmetic operator returns the resulting value.

Arithmetic operators

- The modulus operator requires both variables to be integers.

```
double d1 = 10, d2 = 3;
cout << d1 % d2; // compilation error
```

- The other four operators can take all basic data types.
  - `3 * 8` and `3.2 * 8.4` are all acceptable.
  - This is because these operators have been **overloaded** (for different types).
  - We will discuss **operator overloading** at the end of this semester.

Arithmetic operators

- We need to be particularly careful when doing a division.
- If we use **integers** to be both the dividend **and** divisor, things may go wrong:

```
int d1 = 10;
int d2 = 3;
cout << d1 / d2;
```

  – Try the example "**02_05_intDivision**".
- The result is due to the fact that the operator is defined to **return an integer** if both the operands are integers.
- Solution: storing in a floating point variable or casting.

Example

- Given an integer as a radius, let's calculate the area of that circle.

```
#include <iostream>
using namespace std;

int main()
{
  int radius = 10;
  double area = radius * radius * 3.1416;

  cout << area << "\n";

  return 0;
}
```


Increment/decrement operators

- For many cases, we need to do increment/decrement operations:

```
int i = 10;
i = i + 1; // i becomes 11
i = i - 1; // i becomes 10
```

- In C++, two operators are designed specifically for these tasks.
  – `++`: **increment operator**: `i++` is the same as `i = i + 1`.
  – `--`: **decrement operator**: `i--` is the same as `i = i - 1`.

```
int i = 10;
i++; // i becomes 11
i--; // i becomes 10
```


Increment/decrement operators

- These two operators are both unary.
- Can be applied on all basic data types.
  – But we should only apply them on integers.
- Typically using them is **faster** than using the equivalent addition/subtraction and assignment operation.

Increment/decrement operators

- Both can be put at the **left or** the **right** of the operand.
  - This changes the order of related operations.
  - `i++:` returns the value of `i`, and then increment `i`.
  - `++i:` increments `i`, and then returns the value of `i` after the increment.
  - `i--` and `--i` work in the same way.

```
a = 5; b = a++; // a = 6, b = 5
```

```
a = 5; b = ++a; // a = 6, b = 6
```


Good programming style

- Do not make your program hard to understand.
- What happens to `a = b+++++c`?
- How about `a = (b++) + (++c)`?
- How about

```
c++;
a = b + c;
b++;
```


Precedence

- All operators are ruled by **precedence**.
  - Level 1: `++i`, `--i`.
  - Level 2: `+i`, `-i`, `!i`, `i++`, `i--`.
  - Level 3: `a * b`, `a / b`, `a % b`.
  - Level 4: `a + b`, `a - b`.
  - Level 5: `=`.
- You do not need to remember them, because:
  - They are cumbersome.
  - Even if you can remember all of them, you can not assume other programmer can, too.
- Separate your codes and use parentheses to make your code clear.

The precedence operator

- One may specify the precedence by using **parentheses**: `( )`.
- `{ }` and `[ ]` are not used as parentheses operator.
  - In C++, they have their own meanings.
- For **nested** parentheses, use multiple `()` carefully.

```
a = ((b + c) * d - (e + f)) * g
```

Self-assigning operations

- In many cases, an assignment operation is **self-assigning**.
  - `a = a + b`, `a = a - 20`, etc.
- For each of the five arithmetic operators `+`, `-`, `*`, `/`, and `%`, there is a corresponding **self-assignment operator**.
  - `a += b` means `a = a + b`.
  - `a *= b - 2` means `a = a * (b - 2)` (not `a = a * b - 2`).
- In general, `var op= exp` means `var = var op exp`.
  - `var`: variable. `op`: operator, including: `+`, `-`, `*`, `/`, `%`. `exp`: expression.
- Typically `a += b` is **faster** than `a = a + b`, etc.

Outline

- Basic data types
- Operations, expressions, and statements
- Operators
- **Casting**
- The `cin` object

Casting

- A big container may store a small item.
- A variable of a "larger" type may store a value of a "smaller" type without losing data/precision.

```
double d = 5;   // d = 5.0
int s = 5.5;    // s = 5
```

- There are two kinds of casting:
  - Implicit casting.
  - Explicit casting.

Casting Rules

- Implicit casting:
  - Store a type-1 value to a type-2 variable.
  - Type 2 is "no smaller than" than type 1.
- Examples:
  - `char` -> `int`.
  - `int` -> `float` -> `double`.
  - `short` -> `int` -> `long`.
- Counterexamples:
  - `double` -> `int`, `long` -> `short`.

Casting Rules

- A programmer needs not to ask the compiler to do implicit casting.
  - Because it doesn't cause a loss of precision.
  - The same value may be stored in a different way as the type changes.
- If we want to do something that may cause **a loss in precision**, we should specifically **notify** the compiler.
  - This is the case of explicit casting.
  - This is to make sure that, at the run time, the program runs as we expect.
  - This is to make sure that we know what we are doing.
  - We are also notifying other programmers (or the future ourselves).

Explicit casting

- There are four different explicit casting operators.
  - `static_cast` (the staticCast on p. 111 of the textbook is **wrong**).
  - `dynamic_cast`.
  - `reindivter_cast`.
  - `const_cast`.
- For basic data types, just use **`static_cast`**.

```
static_cast<type>(expression)
```

- For example:
  - `int a = 5.5;` // not good
  - `int a = static_cast<int>(5.5);` // good
- Try the example "**02_06_cast**".

Good programming style

- There is an old way of explicit casting:

```
type (expression)
```

  - For example, `int a = (int) 5.2;` .
- Do not use it!
  - This operation includes all 4 possibilities, and we have no idea which one will be performed.
- If possible, try to modify your variable declaration to avoid casting.

Casting for division

- Recall that if we do

```
int d1 = 10;
int d2 = 3;
cout << d1 / d2;
```

  the result will be 3 rather than 3.33.
- If allowed, we may change the data types of the operands.
  - Try the example "**02_05_intDivision**".
- If not allowed, we may cast the operands temporarily.

Casting for division

- Let's try it:

```
int d1 = 10;
int d2 = 3;
cout << static_cast<double>(d1 / d2);
```

Does that work?

- Let's try another one:

```
int d1 = 10;
int d2 = 3;
cout << static_cast<double>(d1) / d2;
```

Does that work?

Outline

- Basic data types
- Operations, expressions, and statements
- Operators
- Casting
- **The `cin` object**

The `cin` object

- We know that the **`cout`** object can print out data sent into it to the standard input (typically the screen).
- Another object, **`cin`**, can accept data **input by the user** from the standard input (typically the keyboard) into the program.
- In order to use the **`cin`** object, we need to first prepare a **buffer** for the input data. The thing we need is a **variable**.
- When we use a single variable to receive the data, the syntax is

```
cin >> variable;
```

  – The data entered by the user should follow the type of the variable.

The `cin` object

- Consider the following example (example "**02_07_cin**").

```
#include <iostream>
using namespace std;

int main()
{
  int radius = 0;

  cout << "Please enter the radius of a circle: ";
  cin >> radius;

  double area = radius * radius * 3.1416;

  cout << "The area of the circle is " << area << ".\n";

  return 0;
}
```

An example

- An example that really requires explicit casting.
  - The example "**02_08_minutes**".

```
#include <iostream> //
using namespace std;

int main()
{
  double minutes;
  cout << "Enter a number of minutes: ";
  cin >> minutes;

  int minInteger = static_cast<int>(minutes);
  double seconds = (minutes - minInteger) * 60;

  cout << "This converts to " << minInteger
       << " minutes and " << seconds << " seconds.\n";
}
```


The `cin` object

- In this example, we allow the user to enter the radius.
- We define a variable to receive the input value.
- We then use the **`cin`** operation to send the value into the variable.

```
int radius = 0;

cout << "Please enter the radius of a circle: ";
cin >> radius;
```

  - The **`cout`** statement is a **prompt**: a message telling the user what to do.
  - The input of a value ends when the user press "enter".
- The variable can then be used in other statements.

```
double area = radius * radius * 3.1416;
```


The `cin` object

- The **extraction operator** **`>>`** is used with the **`cin`** object.
- One cannot use **`cout`** with **`>>`** or **`cin`** with **`<<`!**

```
a >> cout; // compilation error
b << cin; // compilation error
```

- Multiple variables can be assigned values in a **`cin`** statement.

```
int a = 0;
int b = 0;
cin >> a, b;
```

  - The user may separate the two input values by an "enter" or a white space.
  - Personally I do not recommend this. I will separate the two input actions unless there is a good reason.

Dropped input values

- If in an input stream, there are **more** input values than the variables, values with no corresponding variables will be **dropped**.
- This is particularly an important issue when the user inputs a string.
- As an (not so related) example:

```
char c;
cin >> c; // if we enter "123"
cout << c; // only "1" is printed out

int i, j;
cin >> i >> j; // if we enter "1 2 3"
cout << i << j; // the output is "12"
```

Entering a value with a wrong data type

- The entered value should follow the data type of the variable.
- As an example:

```
int i, j;
cin >> i >> j; // if we enter "1.2 7"
cout << i << j; // the output is ???
```

  - For the **`cin`** object, the decimal point separates two integers: 1 and 2.
  - As the **`cin`** object is expecting two integers, 7 will be dropped.
  - The result is unpredictable.
- What if we change **`i`** to be a **`double`** variable?

Input validation

- In general, it is a **the programmer's responsibility** to avoid potential errors in user input.
  - Users are not programmers/engineers!
- One thing a programmer can do is to provide **clear instructions** in prompts for users.
- Another more important and useful way is **input validation**:
  - Check the data before it is really used.
  - If it is not in the desired type/format, ask the user to re-enter.
  - Will be discussed in the next lecture.
- **Input masks** are widely used in GUI or web programming.
  - Not easy for console inputs.