CSE 143 - Winter 1999

Introduction to C++

[Appendix A]

Introduction to C++

- C++ is a superset of C.
- (Almost) any legal program in C is also a legal C++ program.
- The core of C++ (basic types, variables, operations, and statements) works the same as in C.

Introduction to C++ (cont.)

- Major changes in C++:
  - A "Better C"
  - Support for Data Abstraction (user-defined types)
  - Support for Object-Oriented Programming
- We'll introduce the later two gradually

A Simple C++ Program

```
// A first C++ Program
// Print a greeting message
#include <iostream.h>
int main( ) {
  cout << "Welcome to CSE143!" << endl;
  return 0;
}
```

- **//-**comments extend from **//** to end of line
- Operator **<<** writes the value of the right argument to the output stream on the left, here **cout** - the screen.
- **endl** ends a line of output and ensures that it is displayed Right Now!.

A Second C++ Program

```
// Read two integers and print their sum.
#include <iostream.h>
int main( ) {
  int i, j;
  cout << "Please enter a number: ";
  cin >> i;
  cout << "Please enter another number: ";
  cin >> j;
  cout << "The sum of " << i << " and " << j <<
     " is " << i + j << endl;
  return 0;
}
```


Second C++ Program (cont)

- Operator **>>** reads a value from the stream that is its left argument (here **cin**, the keyboard) and stores it in the variable given as its right argument.
- The **>>** and **<<** operators can be strung together to read or write several items in a single statement.

A "Better C"

- **`cin`** and **`cout`** for stream input and output (plus **`cerr`**)
- Reference parameters
- New comment style
- Relaxed placement of declarations
- Symbolic constants
- A real logical (Boolean) type - **`bool`**
- Enumerated types

Two Styles of Comments

- Old C-style comments

```
/* This is a comment */
```

- Double-slash comments (comment extends from the `//` to the end of the line)

```
int id;    // student ID number
```

- Which form is better?

Declarations Go Anywhere

- C++ declarations can appear anywhere a normal statement can:

```
void something (int x)
{
   if (x == 10)
         x = x / 2;
   int y;  // Declaration can occur here
   ...
}
```

- Common usage: **`for`**-loop index variables

```
for (int k = 0; k < 100; k++) {
   // k is only defined inside this loop
}
```


Symbolic Constants

- Explicit support for constants

```
const double PI = 3.14159;
```

- Do not use **`#define`** …

```
#define PI 3.14159
```

- Why not? Because #define is strictly textual substitution.
- Explicit constants allow compile-time type checking and scope analysis using same rules obeyed by (non-const) variables.

New *bool* type

- *C++ bool* has two legal values: *true* and *false*
  - *bool*, *true* and *false* are reserved words
  - Direct implementation of the "Boolean" concept

```
bool isBigNumber (double d) {
  if (d > 30e6) return true;
  else return false;
}
```

- Not supported in early C++ compilers (one reason you want to have a recent version)

int vs. *bool*

- Under the hood, a **`bool`** value is represented as an **`int`**; **`bool`** and **`int`** values are usually interchangeable (for backward compatibility).
- Use *bool* where Boolean values are natural

```
int i; bool b;
b = (mass >= 10.8); //value is true or false
if (b) ...//OK
while (b && !(i < 15)) ...  //OK
```

- Avoid:

```
i = b;    //marginally OK: value is 0 or 1
i = true; //OK, but bad style
b = i;    //ill-advised (warning)
```

- cout <<
  - displays 0 or 1 for bool values

Enumerated Types

- User-defined type whose constants are meaningful identifiers, not just numbers

```
enum Color { RED, GREEN, BLUE };
```

- Declare like other types; use like other integer types

```
Color skyColor; ...
switch (skyColor) {
  case RED: ...
  case GREEN: ...
  case BLUE: ...
}
```


Input/Output Concepts

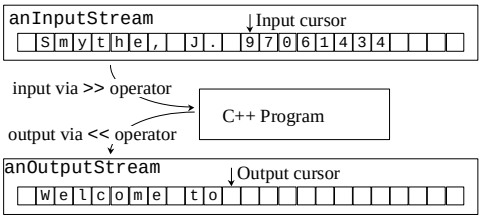
- Concepts should be review!
  - New syntax, but same fundamental concepts
- input vs. output, read vs. write
- conversion between characters in a file and C/C++ data values (types) in a program
- Other concepts that we will return to later:
  - file; file name vs. file variable
  - open; close
  - end-of-file

Stream I/O

- The basic C++ I/O library is built around the concept of streams.
  - both for keyboard/monitor and for files
- Old C-style **`printf`**, **`scanf`**, etc. library is still available, **but**….
  - Mixing the two is bad news
  - You **must** use only stream I/O in CSE143

What is a Stream?

A stream is just a sequence of characters:



Well-Known Streams

- Global streams defined in **`iostream.h`** :
  - **`cin`**: standard *input* stream (usually keyboard)
  - **`cout`**: standard *output* stream (usually screen)
  - **`cerr`**: standard *error* stream (also usually directed to the screen)
- Programs can open other streams to/from files and other devices. We will cover this later, after gaining experience with simple console I/O.

Stream Output Operation

For output streams, **<<** is the “put to” or "insertion" operator

```
#include <iostream.h>
…
int count = 23;
cout << “Hello, World!” << ‘\n’;
          // endl: same as ‘\n’, but flushes output
cout << “The count is ” << count << endl;
```

Stream Input Operation

For input streams, >> is the “get from” or "extraction" operator

```
#include <iostream.h>
…
int x, ID;
char Name[40];
cin >> x;
cin >> Name >> ID;
// Can read multiple items on one line
// Note: no &’s as with scanf
```


How Stream Input Works

Input stream characters are interpreted differently, depending on the data type:

```
int ID;
char Name[40];
char ch;

cin >> ID;   // interprets as integer
cin >> ch;   // reads a char, skipping whitespace
cin >> Name; // interprets as character string,
             // skipping leading whitespace and
             // stopping at trailing whitespace
```


Stream Advantages

- Type Safety
  - No more dangers of mismatched %-types

```
scanf("%s", &someInt);  // oops!
scanf("%d", someInt);   // boom!
```

- Readability
  - Easier to interpret than a sequence of %-types

```
scanf("%d%c%d", &intOne, &oneChar, &intTwo);
versus
cin >> intOne >> oneChar >> intTwo;
```

- Generality
  - Can extend >> and << to work with user-defined data types.

Parameters

- What does this print?

```
#include <iostream.h>
…
// Double the value of k
void dbl(int k) { k = 2 * k; }

int main( ) {
  int n = 21;
  dbl(n);
  cout << n << endl;
}
```

- Output:

Reference Parameters

- Use & in parameter declaration to make the parameter an alias for the argument.

```
// Double the value of k
void dbl(int & k) { k = 2 * k; }

int main( ) {
  int n = 21;
  dbl(n);
  cout << n << endl;
}
```

- Output:

C++ Parameters

- Value parameters (as in C):
  - Passes a *copy* of the actual argument (except arrays)
  - Assignment to parameter in function **doesn’t** change actual arguments (except arrays)
- Reference parameters:
  - The parameter is an **alias** for actual argument
  - Achieves same effect as pointer parameters without needing explicit use of **&** and ***.**
  - Assignments to parameter changes argument.
- We will use pointers in data structures later. Avoid them for now.