

## CSE 143

### Introduction to C++

[Appendix A]

9/27/00 B-1

## C++ vs. C

- C++ is a superset of C
  - C++ has a huge number of new features
  - Often criticized as overly complex
- (Almost) any legal program in C is also a legal C++ program.
- The core of C++ works the same as in C
  - basic types, variables, expressions
  - declaring and using functions
  - statements (if, while, for, etc.)

9/27/00 B-2

## C++ vs. 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 latter two gradually
- Today we focus on some of the "better C" features

9/27/00 B-3

## A Simple C++ Program

```
// A first C++ Program
// Print a greeting message
#include <iostream>
using namespace std;
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!.

9/27/00 B-4

## A Second C++ Program

```
// Read two integers and print their sum.
#include <iostream>
using namespace std;
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;" is assumed at the end of main if it is omitted.

9/27/00 B-5

## 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.
- **Important:** Place your C++ functions in .cpp files (rather than .c files).
- The "using namespace std;" directive accesses the names defined in the header files

9/27/00 B-6

## A “Better C”

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

9/27/00 B-7

## 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?

9/27/00 B-8

## 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
}
```

9/27/00 B-9

## Symbolic Constants

- Explicit support for constant variables  
`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.
- More about `const` another day

9/27/00 B-10

## 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)

9/27/00 B-11

## *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

9/27/00 B-12

## 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 values

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

9/27/00 B-13

## Structs as Types

- Old way:

```
typedef struct {
    ...
} Student_record;
```
- New way:

```
struct Student_record {
    ...
};
```
- Convention: New type names are capitalized
- Preview: in C++ we often use *class* instead of *struct*  
Almost identical, but different implications

9/27/00 B-14

## Defining 'main'

- Usual signature for main (as in C, except void keyword not needed as a parameter):

```
int main ( ) { ... return x; ... }
```
- Sometimes seen in old code (but nonstandard!):

```
void main ( ) { .... /*no return*/ ... }
```
- A few others are possible, too. For you hackers:

```
int main (int numArgs, char * argArray[ ]);
```

  - Allows OS to provide command line arguments to the program.

9/27/00 B-15

## Parameters (Review)

- Puzzler: What does this print?

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

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

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

- Output:

9/27/00 B-16

## Reference Parameters (1)

- The default in C/C++ is pass by *value*
  - a copy of the actual argument is made
  - exception: arrays
- C technique for passing by *reference*: pass a pointer to the argument
  - Can still do this in C++
- Passing by ref. is more efficient for large objects -- why?
- Passing by ref. is less safe than by value -- why?  
*C++ offers an additional technique...*

9/27/00 B-17

## Reference Parameters (2)

- 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:

9/27/00 B-18

## Reference Parameters (3)

- The parameter is an **alias** for actual argument
- Achieves same effect as pointer parameters
  - & when parameter declared
  - No explicit & in argument
  - No explicit \* when parameter used
- Assignments to parameter changes argument
  - Why? because one is an alias of the other

9/27/00 B-19