



C++ Basics

Brian Malloy, PhD
Department of Computer Science
Clemson University
Clemson SC, USA

C++ Overview

- Designed by B. Stroustrup (1986),
- C++ and ANSI C
- Hybrid language: OO and 'conventional' programming,
- More than just an OO version of C
 - operator overloading
 - templates

2

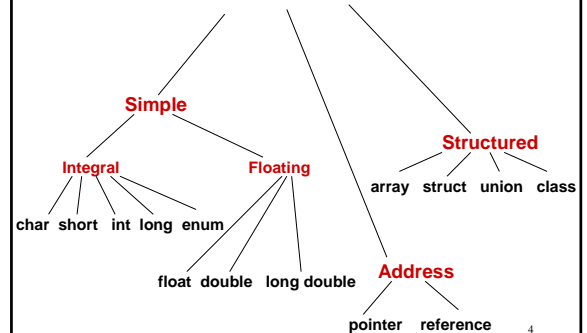
Identifiers

- An identifier must start with a letter or underscore, and be followed by zero or more letters
- **C++ is case sensitive**
- **VALID**

<code>age_of_dog</code>	<code>TaxRate98</code>
<code>PrintHeading</code>	<code>AgeOfHorse</code>

3

C++ Data Types



4

Enumerated types: *enum*

- Used to define constant values
- ```

enum days
{ Sunday = 0, Monday, Tuesday,
 Wednesday, Thursday, Friday,
 Saturday
} yesterday, today;

```

5

## Boolean type

- C++ has a built-in logical or Boolean type
- ```

bool flag = true;
  
```

6

Named Constants

- A **named constant** is a location in memory which we can refer to by an identifier, and in which a **data value that cannot be changed** is stored.

VALID CONSTANT DECLARATIONS

```
const char STAR = '*';
const float PI = 3.14159;
const int MAX_SIZE = 50;
```

Notes: all caps

7

some keywords: reserved to C++

- bool, true, false,
- char, float, int, unsigned, double, long
- if, else, for, while, switch, case, break,
- class, public, private, protected, new, delete, template, this, virtual,
- try, catch, throw.

frequently used keywords

8

two portable versions of main

type of returned value name of function says no parameters

```
int main ( ) {
    return 0;
}
```

the return statement is optional

main returns an integer to the operating system

9

two portable versions of main

```
int main (int argc, char * argv[] )
{
    return 0;
}
```

the return statement is optional

main returns an integer to the operating system

10

Prefix vs postfix

- When the increment (or decrement) operator is used in a statement with other operators, the prefix and postfix forms can yield **different** results.

```
int x = 3, y = 3;
cout << ++x << endl;
cout << y++ << endl;
```

what's the output?

11

Operators can be

binary involving 2 operands $2 + 3$

unary involving 1 operand $- 3$

ternary involving 3 operands $?:$

12

CONTROL STRUCTURES

Use logical expressions which may include:

6 Relational Operators

< <= > >= == !=

3 Logical Operators

! && ||

19

Short-circuit

```
int Number;  
float X;
```

```
( Number != 0 ) && ( X < 1 / Number )
```

20

Compound statement

- We use braces {} to build compound - statements
- To use braces or not to use braces??

```
for (i = 0; i < n; ++i)  
    sum += i;
```

```
for (i = 0; i < n; ++i) {  
    sum += i;  
}
```

21

Iteration statements

```
// compute sum = 1 + 2 + ... + n
```

```
int sum = 0;  
i = 1;  
while (i <= n) {  
    sum += i;  
    i++;  
}
```

init of the lcv
loop termination condition.
body of the loop
incr of the lcv

22

Iteration statements

```
// compute sum = 1 + 2 + ... + n
```

```
int sum = 0;  
for (int i = 1; i <= n; ++i) {  
    sum += i;  
}
```

i doesn't exist here!

23

Combining *break* and *for*

```
char ch;  
int count = 0;
```

```
for (; ; ) {  
    cin >> ch;  
    if (ch == '\n') break;  
    ++count;  
}
```

Alternatives?

24

Switch

```
switch (letter) {
  case 'N': case 'n': cout < "New York\n";
              break;
  case 'L': case 'l': cout < "London\n";
              break;
  case 'A': case 'a': cout < "Amsterdam\n";
              break;
  default:   cout < "Somewhere else\n";
              break;
}
```

25

What's the output?

```
int i = 0;
switch ( i ) {
  case 0 : i+= 5;
  case 1 : ++i;
  case 2 : ++i;
  case 3 : ++i;
  default ++i;
}
cout << "i is: " << i << endl;
```

26

Conditional expression

```
// example of the conditional expression

int a = 7, b = 9;
cout << "The greater of a and b is"
     << (a > b ? a : b);
```

What's the output?

27

Simple arrays

- subscripts can be an integer expression
- In the declaration, the dimension must be a constant expression

```
const int LENGTH = 100;
...
int a[LENGTH]
...
for (int i=0; i<LENGTH; i++)
  a[i] = 0; // initialize array
```

known at compile time!

28

Type safety

- Every arithmetic expression has a type
- This type can be derived from those of its operands
- Try to be type-safe: **Don't mix types!**

29

Functions: 3 parameter transmission modes

- pass by *value* (default) local copy
- pass by *reference* (&) pass the address
- pass by *const reference* (const &) good for big structures



30

Functions: example of pass by value

```
int sqr(int x) {  
  
}
```

The compiler makes
a local copy!

31

Functions: example of pass by reference

```
void swap(int &x, int &y) {  
  
}
```

Must pass by reference!

32

Functions: pass by *const reference*

- Makes sense with large structures or objects

We'll use it when
we make objects.

33

Arrays are passed by reference

```
const int MAX = 100;  
void init(int a[], int x) {  
    for (int i = 0; i < MAX; ++i) {  
        a[i] = rand() % 100;  
    }  
    x = 99;  
}  
main() {  
    int a[MAX], x = 7;  
    init(a, x);  
    cout << a[0] << '\t' << x << endl;  
}
```

34

The void keyword

- In C one might write

```
main()
```

- This is equivalent to:

```
int main()
```

35

Functions: initialization

```
#include <iostream>  
  
void f() {  
    static int i=1;  
    cout << i++ << endl;  
}  
  
int main() {  
    f();  
    f();  
    return 0;  
}
```

What's the output?

36

A static variable can be used as a flag

```
void f() {
    static bool first_time = true;
    if (first_time) {
        cout << "f called for the first time\n";
        first_time = false; // false
    }
    cout << "f called (every time)\n";
}
```

static constant can also
be used in classes

37

Functions: initialization

```
void f(int i, float x=0, char ch='A')
{
    ...
}
...
f(5, 1.23, 'E');
f(5, 1.23); // equivalent to f(5,1.23,'A');
f(5); // equivalent to f(5,0,'A');
```

38

Function overloading

- two or more functions with the same name
- The number or types of parameters must differ:

```
void writenum(int i) {
    cout << "i is " << i << endl;
}

void writenum(float x) {
    cout << "x is: " << x << endl;
}
```

39

Functions: References as return values

- A value can be returned from a function using any of the 3 transmission modes.
- This is especially important when passing objects.

Pass by value makes a copy!

40

constants in a class:

```
class Game {
private:
    static const int NUM_TURNS;
    int scores[NUM_TURNS];
};
```

declaration

```
const int Game::NUM_TURNS = 5;
```

definition

41

Which is better printf or cout?

```
#include <iostream>
```

```
int i;
Rational r;
```

```
cout << i;
cout << r;
```

42

What's the difference between `<iostream.h>` and `<iostream>`

- `<iostream>` is part of the ANSI standard C++ library; it's in `std` namespace
- `<iostream.h>` is an artifact of pre-standard C++; it's not in `std`
- However, vendors do not want old code to break; thus, `<iostream.h>` will likely continue to be supported

43

What's the difference between `string.h` and `string`

- similar to `iostream.h` and `iostream`
- `<string.h>` -- the old C-string (`char *`)
- `<string>` -- C++ string and C-string
- `<cstring>` -- old C-string in `std` namespace

44

C strings:

- C++ has them but: artifact of C
- *programmer must manage memory*
- terminated with `'\0'`
- built-in c-string library:
 - `strlen(s)`: returns length of string
 - `strcat(s, t)`: place `t` at end of `s`
 - `strcpy(s, t)`: copy `t` into buffer `s`
 - `strcmp(s, t)`: 0 if `s==t`, <0 if `s<t`, >0 if `s>t`

45

Command line parameters

- Integer parameters must be converted:

```
main(int argc, char * argv[]) {  
    if (argc < 2) {  
        cout << "usage: " << argv[0]  
            << " <number>" << endl;  
        return 1;  
    }  
    int n = atoi(argv[1]);  
}
```

46

C++ & Object-Oriented Programming



The Class



47