Basics: From C to C++ (2)

Computer Programming for Engineers (DASF003-41)

Instructor:

Sungjae Hwang

- jason.sungjae.hwang@gmail.com
- https://softsec-lab.github.io/

Calendar

Tentative Schedule

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
W1	28	29	30	31	9/1	2	3
W2	4	5	6	7	8	9	10
W3	11	12(대체 휴일)	13	14	15	16	17
W4	18	19	20	21	22	23	24
W5	25	26	27	28 (PA1)	29	30	10/1
W6	2	3(개천절)	4	5	6	7	8
W7	9	10(대체 휴일)	11	12	13	14	15
W8	16	17	18	19(midterm)	20	21	22
W9	23	24	25	26 (PA2)	27	28	29
W10	30	31	11/1	2	3	4	5
W11	6	7	8	9	10	11	12
W12	13	14	15	16 (PA3)	17	18	19
W13	20	21	22	23	24	25	26
W14	27	28	29	30	12/1	2	3
W15	4	5	6	7(final)	8	9	10

Goal

- **Textbook : Absolute C++ 6th edition (Walter Savitch)**

  - Type Casting (1.2)
  - Control Flow (2)
  - Structured Binding (External Materials)
    - https://en.cppreference.com/w/cpp/language/structured_binding
    - https://www.geeksforgeeks.org/structured-binding-c/

TYPE CASTING

C-Style Type Casting (still accepted in C++)

- **Two types**
  - **Implicit** - also called “Automatic”
    - Done FOR you, automatically

```
17 / 5.5
```

    - This causes an “implicit type cast to take place, casting the 17 -> 17.0”
  - **Explicit** type conversion
    - Programmer specifies conversion with cast operator
    - C style / older form : **(type) expression;**

```
// same expression as above, using explicit cast
(double) 17 / 5.5
// more typical use; cast notation on variable
(double) myInt / myDouble
```

C Style Type Casting

```
#include <iostream>

using namespace std;

int main()
{
  int intVar1 = 1, intVar2 = 2;
  cout << 1 / 2 << endl;
  cout << intVar1 / intVar2 << endl;
  cout << 1.0 / 2 << endl;
  cout << 1 / 2.0 << endl;

  // following line doesn't work
  //cout << intVar1.0 / intVar1 << endl;
  //               ~~
}
```

C++ Style Type Casting

- Two types
  - Explicit type conversion (C++ style casting)
    - Actually, the following style is preferred in C++
    - *type(expression);*

```
// same expression as above, using explicit cast
// (double) 17/5.5
double(17)/5.5
// more typical use;
double(myInt) / myDouble
```

C++ Style Type Casting

```
#include <iostream>

using namespace std;

int main()
{
  cout << endl << "diffent styles of casting" << endl;
  int myInt = 1;
  double myDouble = 2;
  cout << (double) 1/2 << endl;
  cout << (double) (1/2) << endl;
  cout << 1/(double)2 << endl;
  cout << double(1)/2 << endl;
  cout << myInt/myDouble << endl;
  cout << double(myInt)/myDouble << endl;
}
```

Four Type Casting Operators in C++

- **static_cast<>()**
  - Similar to the simple C-style casting
- **const_cast<>()**
  - Getting a write access to something declared const
- **reinterpret_cast<>()**
  - Will be explained later (after learning class)
- **dynamic_cast<>()**
  - Related to polymorphism, and will be explained later

Type Casting Operators in C++

- **static_cast**
  - Can add “.0” to literals to force precision arithmetic, but what about variables? We can’t use “myInt.0”!
  - Explicitly “casts” or “converts” intVar1 to double type
  - *static_cast<type>(expression)*

```
double doubleVar = static_cast<double>(intVar1) / intVar2;
```

- **Why static_cast rather than C style casting?**
  - Compiler can check the correctness.

static_cast

DEMO

```
#include <iostream>

using namespace std;

int main()
{
  // static_cast
  int intVar1 = 1, intVar2 = 2;
  cout << endl << "static_cast example" << endl;
  double doubleVar = static_cast<double>(intVar1) / intVar2;
  cout << doubleVar << endl;;
  doubleVar = intVar1 / static_cast<double>(intVar2);
  cout << doubleVar << endl;;
  doubleVar = static_cast<double>(intVar1 / intVar2);
  cout << doubleVar << endl;;
}
```

Type Casting Operators in C++

const_cast

- We can remove **const** modifier using **const_cast** for references and pointers
- *const_cast<type>(expression)*

```
double var1;
const double& var2 = var1;
double& var3 = const_cast<double&>(var2);
```

- Note that we use **double&** instead of double
- **&** is a reference:
  - Var3 now refers to var2, where modifying var3 also affects var2 and var1.

const_cast

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

int main(){
  //ex1
  const double a = 1.1;
  double b = const_cast<double>(a);

  //ex2
  //double a = 1.1;
  //const double& b = a;
  //b = 2.2;
  /*
  double& c = const_cast<double&>(b);
  c = 2.2;
  cout << "a: " << a << ", b: " << b << ", c: " << c << endl;
 */


```

const_cast

DEMO

```
  //ex3
  //Beware! It is undefined behavior to modify a value which is initially declared as const.
  /*
  const double a = 1.1;
  const double& b = a;
  double& c = const_cast<double&>(b);
  c = 2.2;
  cout << "a: " << a << ", b: " << b << ", c: " << c << endl;
*/
  return 0;
}
```

CONTROL FLOW

Control Flow

- **Most of C control flow still applies the same to C++.**
  - if-else, for/while/do-while loops, switch, ternary operator (?:)

- **We just skip such basic stuff, here.**

Range-based for loop (C++11)

- The C++ ranged-based for loop makes it easy to iterate over each element in a loop
- Format

```
for (datatype varname : array)        {
//  varname is set to each successive element in the array
}
```

- Example

```
int arr[] = {20, 30, 40, 50};
for( auto x : arr ) cout << x << " ";
cout << endl;
```

```
20 30 40 50
```

Ranged for loop

DEMO

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

//example of range-based for loop
int main()
{
  //ex1
  int arr[] = {20, 30, 40, 50};

  for(int i=0; i<sizeof(arr)/sizeof(int); i++) // messy :(
    cout << arr[i] << " ";
  cout << endl;

  for( auto x : arr ) // beautiful :)
    cout << x << " ";
  cout << endl;

  //ex2
  //string str = "abcd";
  char* str = "abcd";
  for( auto c : str)
    cout << c << endl;
```

Range-based for loop (C++11)

- Pass-by-value (can’t change the value)

```
int arr[] = {20, 30, 40, 50};
for( auto x : arr ) x++;
for( auto x : arr ) cout << x << " ";
cout << endl;
```

```
20 30 40 50
```

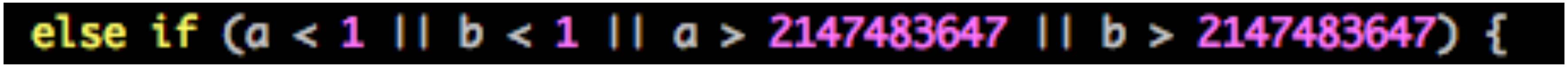
- Pass-by-reference (can change the value)

```
int arr[] = {20, 30, 40, 50};
for( auto& x : arr ) x++;
for( auto x : arr ) cout << x << " ";
cout << endl;
```

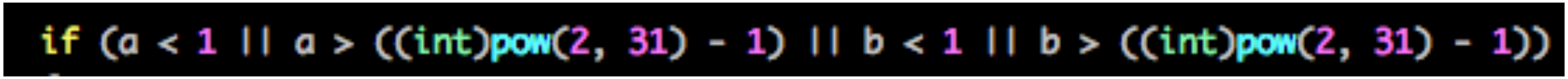
```
21 31 41 51
```

Pitfalls #1

- Check the integer is ranging from 1 to 2147483647 (=2^31-1)



```
else if (a < 1 || b < 1 || a > 2147483647 || b > 2147483647) {
```



```
 if (a < 1 || a > ((int)pow(2, 31) - 1) || b < 1 || b > ((int)pow(2, 31) - 1))
```

Casting

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

int main() {
           //2147483647 vs 2147483648 vs 2147483646
  cout << "test 1:" << ((int)pow(2, 31) - 1) << endl; //student code
           cout << "test 2:" << ((long)pow(2, 31) - 1) << endl;
           cout << "test 3:" << ((int)(pow(2, 31) - 1)) << endl;

           int a,b;
           cin >> a;
           cin >> b;
    //check range 1~2147483647
    if (a < 1 || a >((int)pow(2, 31) - 1) || b < 1 || b >((int)pow(2, 31) - 1))
    {
      cout << "Min: Max: Error: out of range";
    }
           return 0;
}
```

Pitfalls #2

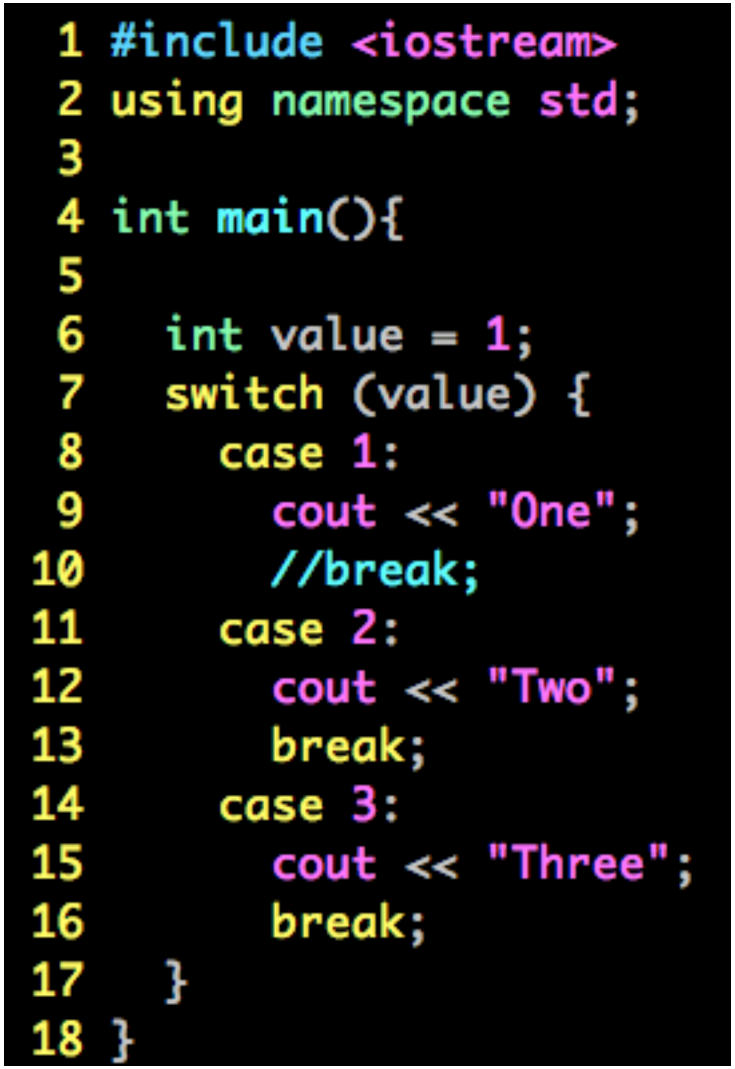
- Integer Overflow in for loop

Pitfalls #3

- **Short-circuit evaluation**
  - (x >= 0) && (y > 1)
  - Be careful with increment operator!
    - (x > 1) && (y++)

Pitfalls #4

- **Switch pitfalls**
  - Forgetting the break;
    - No compiler error



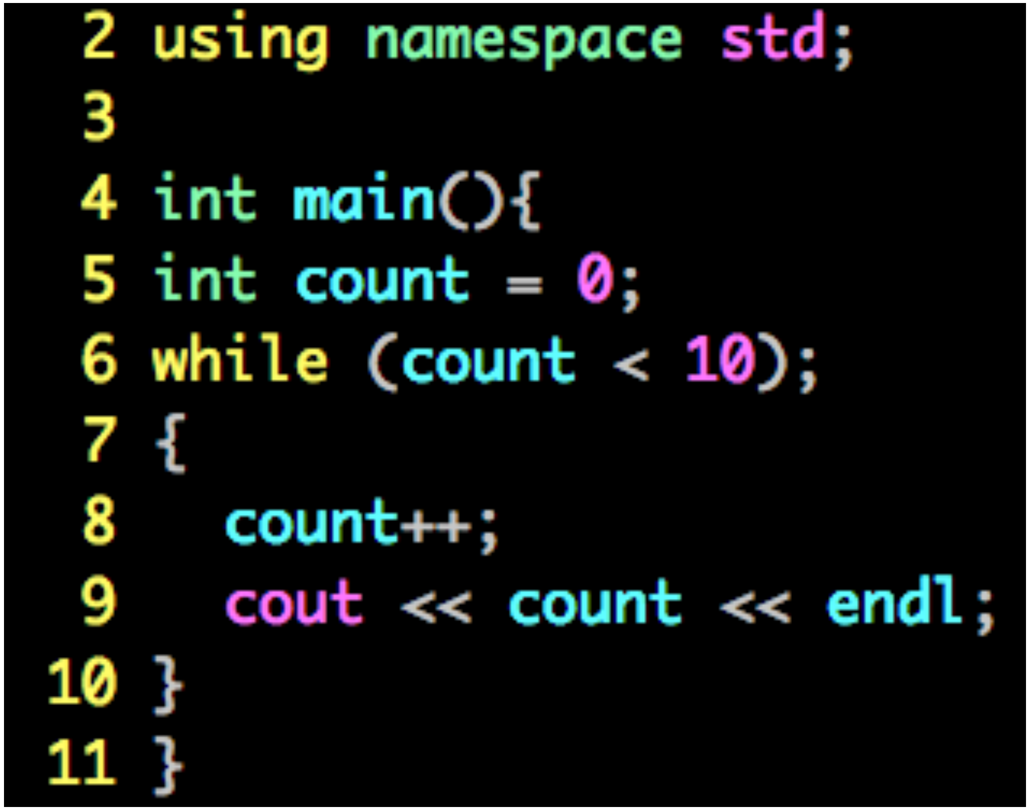
```
#include <iostream>
using namespace std;

int main(){

  int value = 1;
  switch (value) {
    case 1:
      cout << "One";
      //break;
    case 2:
      cout << "Two";
      break;
    case 3:
      cout << "Three";
      break;
  }
}
```

Pitfalls #5

- Loop pitfalls

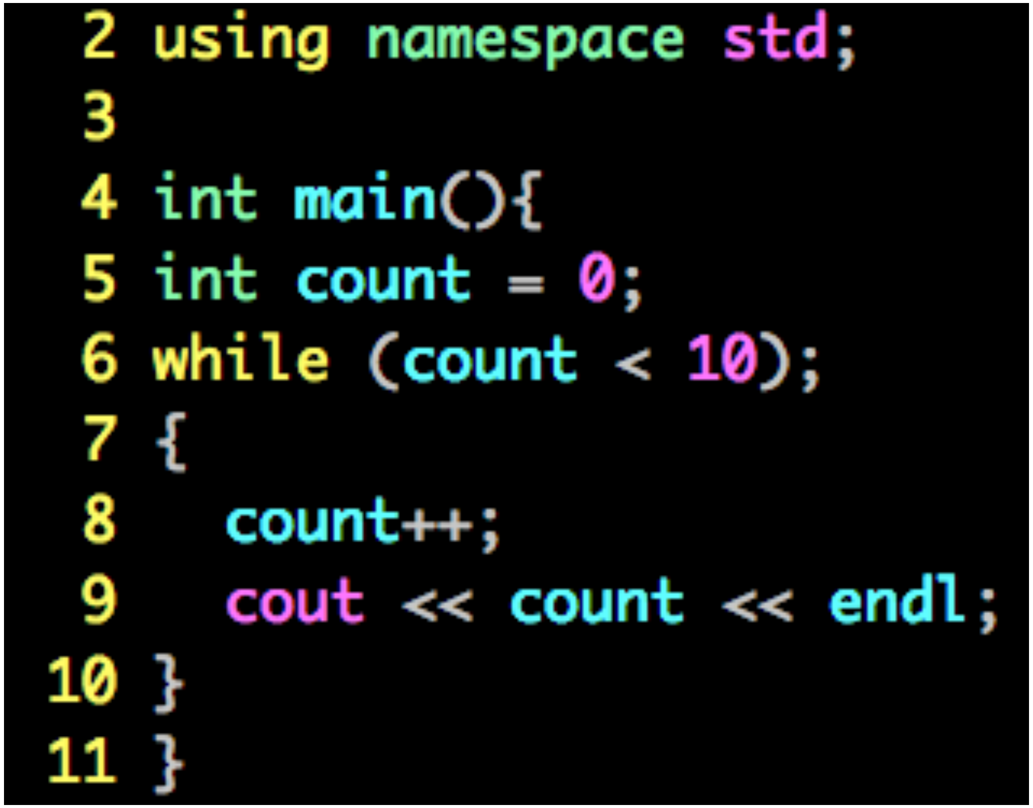


```
using namespace std;

int main(){
int count = 0;
while (count < 10);
{
   count++;
   cout << count << endl;
}
}
```

Pitfalls #6

- **Loop pitfalls**
  - Watch the misplaced ;(semicolon)



```
using namespace std;

int main(){
int count = 0;
while (count < 10);
{
  count++;
  cout << count << endl;
}
}
```

STRUCTURED BINDING (C++17)

Structured Binding

- **Latest C++ (since C++17) allows us to batch-assign multiple variables using auto.**
  - This works for an array, structure members, tuple, and STL iterators
  - *Auto ref-operator(optional)[ids] = expressions;*

- **Examples**
  - Binding an array

```
int a[2] = {1,2};
auto [x,y] = a;
auto& [xr, yr] = a; // xr/yr refer to a[0]/
```

  - Binding a structure

```
struct { int i=1; double d=2; } f;
auto [x, y] = f;
std::cout << x << " " << y << std::endl; // 1
```

Structured Binding

```
#include <iostream>

using namespace std;

int main()
{
#if 1
  int a[2] = {1,2};
  auto [x,y] = a;
  auto& [xr, yr] = a;
  cout << x << "," << y << endl;

  xr = 3;
  yr = 4;

  // what will be the result?
  cout << x << "," << y << endl;
  cout << a[0] << "," << a[1] << endl;
  cout << xr << "," << yr << endl;
#else
  struct {
    int i=1;
    double d=2;
  } f;
  auto [i,d] = f;
  cout << i << " " << d << endl;
#endif

  return 0;
}
```