Basics: From C to C++

Computer Programming for Engineers (DSAF003-42)

Instructor:

Youngjoong Ko (nlp.skku.edu)

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 (C style type casting)
    - Programmer specifies conversion with cast operator

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

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

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

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

C style 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;

}
```

C++ style 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;
}
```

CONSTANTS AND ENUMERATION

Named Constants in C++

- **Naming your constants**
  - Literal constants (e.g., 24) are "OK", but provide little meaning
    - e.g., 24 tells nothing about what it represents.

- **Use named constants instead**
  - Meaningful name to represent data
  - In C, constants are often declared as macros

```
#define NUM_STUDENTS   24
#define NULL           0
```

  - Now, we know the meaning, but type is still ambiguous.
    - e.g., Is NULL zero integer? or null pointer? or double?

Named Constants in C++

- **Typed named constants**
  - Meaningful name to represent data

```
const int NUM_STUDENTS = 24;
```

  - Called a "declared constant" or "named constant."
  - Now use its name wherever needed in program.
  - Added benefit: changes to value result in one fix.

Named Constants in C++

- **Named Constant**

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

int main( )
{
    const double RATE = 6.9;
    double deposit;
    cout << "Enter the amount of your deposit $";
    cin >> deposit;
    double newBalance;
    newBalance = deposit + deposit*(RATE/100);

    cout << "In one year, that deposit will grow to\n"
    << "$" << newBalance << " an amount worth waiting for.\n";
    return 0;
}
```

```
Enter the amount of your deposit $100
In one year, that deposit will grow to
$106.9 an amount worth waiting for.
```

enum

- **enum can be used to systematically declare multiple (having sequential values) constants.**

```
enum MODE { WEAPON, EQUIPMENT, GEM = 10, DEFENSE, };
```

  - each item type in enum is assumed to be an integer.

enum


```
#include <iostream>
using namespace std;
int main() {
  enum MODE { WEAPON, EQUIPMENT, GEM,  DEFENSE};
  int mode;
  cout << "Enter mode(0:Weapon, 1:Equipment, 2:Gem, 3:Defence): ";
  cin >> mode;
  switch(mode) {
  case WEAPON:
    cout << "Weapon" << endl; break;
  case EQUIPMENT:
    cout << "Equipment" << endl; break;
  case GEM:
    cout << "Gem" << endl; break;
  case DEFENSE:
    cout << "Defence" << endl; break;
  default:
    cout << "Wrong mode" << endl;
  }
}
```

Strong enum (C++11)

- **C++11 introduces strong enums or enum classes**
  - Does not act like an integer.
  - Examples

```
enum class Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
enum class Weather { Rain, Sun };

Days d = Days::Tue;
Weather w = Weather::Sun;
```

  - Illegal: `if (d == 0)`
  - Legal: `if (d == Days::Wed)`

Strong enum


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

int main() {
  enum class IOResult {Error, Ok};
  enum class ParseResult {Error, Ok};
  IOResult io_return_code = IOResult::Ok;
  switch(io_return_code) {
    case IOResult::Ok:
      cout << "IO done" << endl; break;
    case IOResult::Error:
      cout << "IO Error" << endl;
  }
  ParseResult parse_return_code = ParseResult::Error;
  switch(parse_return_code) {
    case ParseResult::Ok:
      cout << "Parse done" << endl; break;
    case ParseResult::Error:
      cout << "Parse Error" << endl;
  }
}
```

CONTROL FLOW

Control Flow

- **Most of C control flow still applies the same to C++.**
  - if-else (conditional statement), for/while/do-while loops, switch, ternary operator (?:)
- **We just skip such basic stuff, here.**

Range-based for loop (C++11)

- The C++11 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
```

for


```
#include <iostream>

using namespace std;

int main()
{
  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;

  string str = "abcd";
  //Check with the below line
  //char* str = "abcd";
  for( auto c : str)
    cout << c << 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.
  - Tuple and STL iterators are not covered yet.
- **Examples**
  - Binding an array

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

  - Binding a structure

```
struct { int i=1; double d=2; } f;
auto [x, y] = f;
std::cout << x << " " << y << std::endl; // 1 2.000000
```

structured binding


```
#include <iostream>

using namespace std;

int main()
{
  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;

  struct {
    int i=1;
    double d=2;
  } f;
  auto [i,d] = f;
  cout << i << " " << d << endl;

  return 0;
}
```

POINTERS &
DYNAMIC MEMORY ALLOCATION

nullptr (C++11)

- **Ambiguity of NULL in C**
  - no distinction between integer 0 and null pointer

```
void func( int* p );
void func( int i );
```

  - Which func is invoked given `func(NULL)`? Both are equally valid since NULL is merely 0.
- **C++11 resolved this problem by introducing nullptr**

```
int* p_int = nullptr;
```

  - the type of `nullptr` is `std::nullptr_t`

nullptr


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

int main(void) {
     // NULL, nullptr 비교2
    cout << endl << "== NULL, nullptr 비교2" << endl;
    int* ptr1 = NULL;
    int* ptr2 = nullptr;

    if (ptr1 == NULL) { cout << "2-1. NULL == NULL" << endl; }
    if (ptr2 == NULL) { cout << "2-2. nullptr == NULL" << endl; }
    if (ptr1 == nullptr) { cout << "2-3. NULL == nullptr" << endl; }
    if (ptr2 == nullptr) { cout << "2-4. nullptr == nullptr" << endl; }
    if (ptr1 == ptr2) { cout << "2-5. NULL == nullptr" << endl; }

    // NULL, nullptr 비교3
    cout << endl << "== NULL, nullptr 비교3" << endl;
    int a = 0;
    if (a == NULL) { cout << "3-1. int 타입 0 == NULL" << endl; }
    //ERROR if (a == nullptr) { cout << "3-2. int 타입 0 == nullptr" << endl; }

    return 0;
}
```

malloc and free (in C)

- **malloc()/free() functions can be used to allocate/deallocate memory.**
  - e.g., dynamically allocate/release a single integer

```
int* ptr = (int*) malloc( sizeof(int)*1 );
if(ptr) free( ptr );
ptr = NULL;
```

  - e.g., dynamically allocate/release 10 doubles

```
double* ptr = (double*) malloc( sizeof(double)* 10 );
if(ptr) free( ptr );
ptr = NULL;
```

new and delete operators (C++)

- **new/delete can replace malloc/free with typed sizes.**
  - e.g., dynamically allocate/release a single integer

```
int* ptr = new int;
if(ptr!=nullptr) delete ptr;
ptr=nullptr;
```

- **new[]/delete[] for arrays**
  - e.g., dynamically allocate/release 10 doubles

```
double* ptr = new double[10];
if(ptr!=nullptr) delete[] ptr;
ptr=nullptr;
```

    - delete[] indicates the ptr is an array.

More on new and delete operators (C++)

- **POD types**
  - the results of new/delete performs exactly the same as malloc/free.

- **Classes/Structures**
  - new = allocation + invocation of constructor
  - delete = invocation of destructor + deallocation
  - constructors/destructors will be explained later.

new, delete


```
#include <iostream>

using namespace std;

int main()
{
  int* ptr = new int;
  *ptr = 1;
  cout << *ptr << endl;

  // guess results
  cout << ptr << endl;
  cout << sizeof(ptr) << endl;
  cout << sizeof(*ptr) << endl;

  if(ptr != nullptr) delete ptr;
  ptr = nullptr;
  return 0;
}
```