Basics: From C to C++

Computer Programming for Engineers (DSAF003-42)

Fall, 2021

Instructor:

Youngjoong Ko (nlp.skku.edu)

Announcement - Lab Sessions (20%)

- Time
  - 12:00 pm ~ 1: 15 pm, Wednesday
  - It is a kind of hands-on class (practical class)
- Weekly assignments
  - For checking your programming skill improvement
- Getting points
  - You should submit the weekly programming assignments
    (You can discuss the problems with the TAs).
  - Deadline: **11:59 pm on Fridays**
  - If you miss the deadline, you do not have a chance to submit them.
  - If you submit only a subset of the problems or submit wrong answers, you cannot get a credit.
  - You MUST upload a submitted file to iCampus for double checking.

Announcement about Demo

- All the source codes for demo will be shared on website (github) of NLP lab.
  - https://nlplab.skku.edu (find link to github)
  - https://github.com/NLPlab-skku/

Basics: From C to C++

Computer Programming for Engineers (DSAF003-42)

Fall, 2021

Instructor:

Youngjoong Ko (nlp.skku.edu)

Intro.

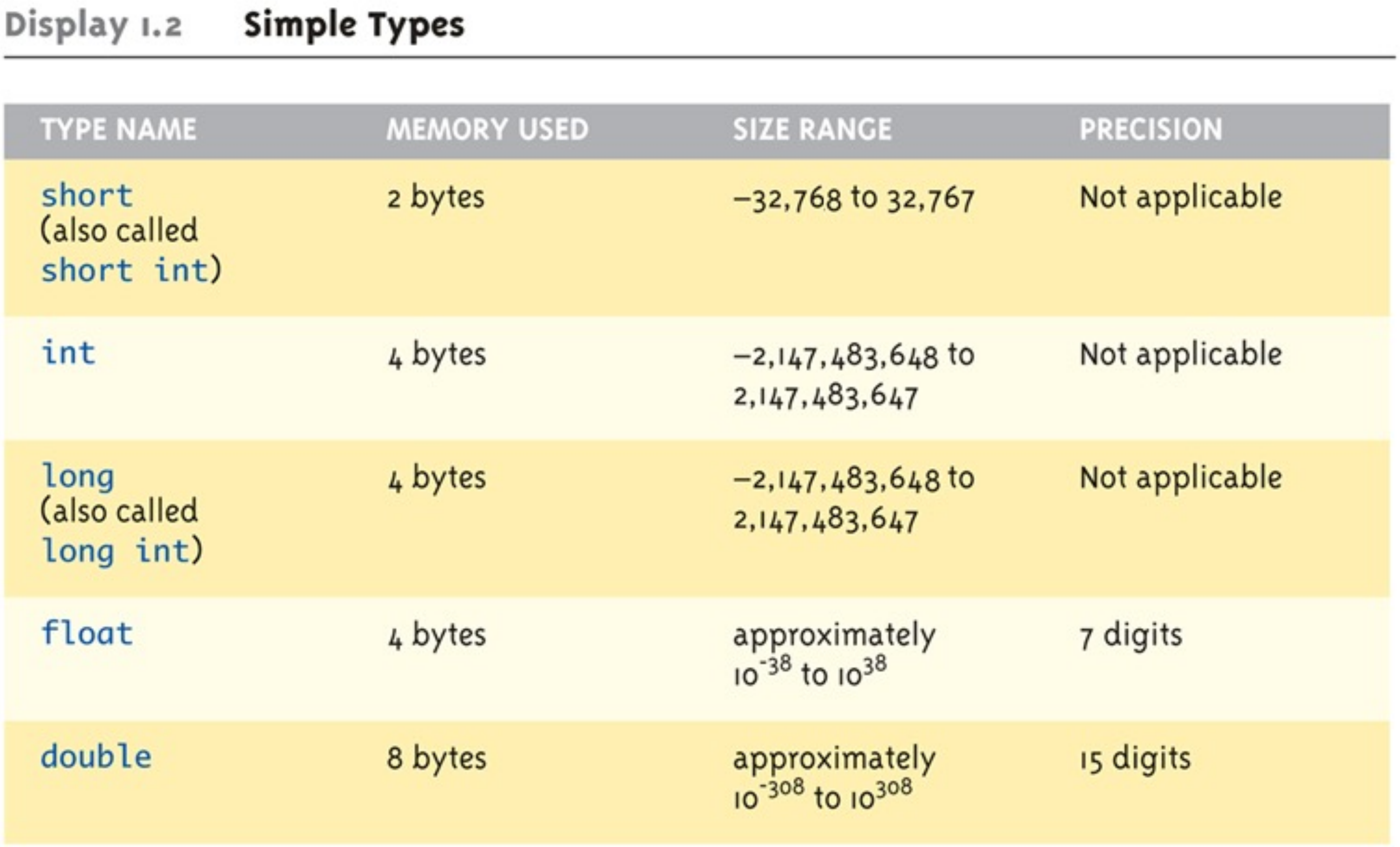
- C++ is a superset of C.
  - This means you can try anything you can use in C.
  - All the basic types and control flow of C is accepted in C++.

- C++ extends C in better/advanced ways.
  - In this lecture, we will investigate many different/extended aspects of C++.
  - This material marks modern C++11 for some items; the others are typically a part of C++98 standard.

- In case you are not familiar with C,
  - Please intensively review the C programming immediately.
  - In this course, I will always presume you are familiar with C programming.

TYPES

Plain Old Data (POD) Types in C/C++

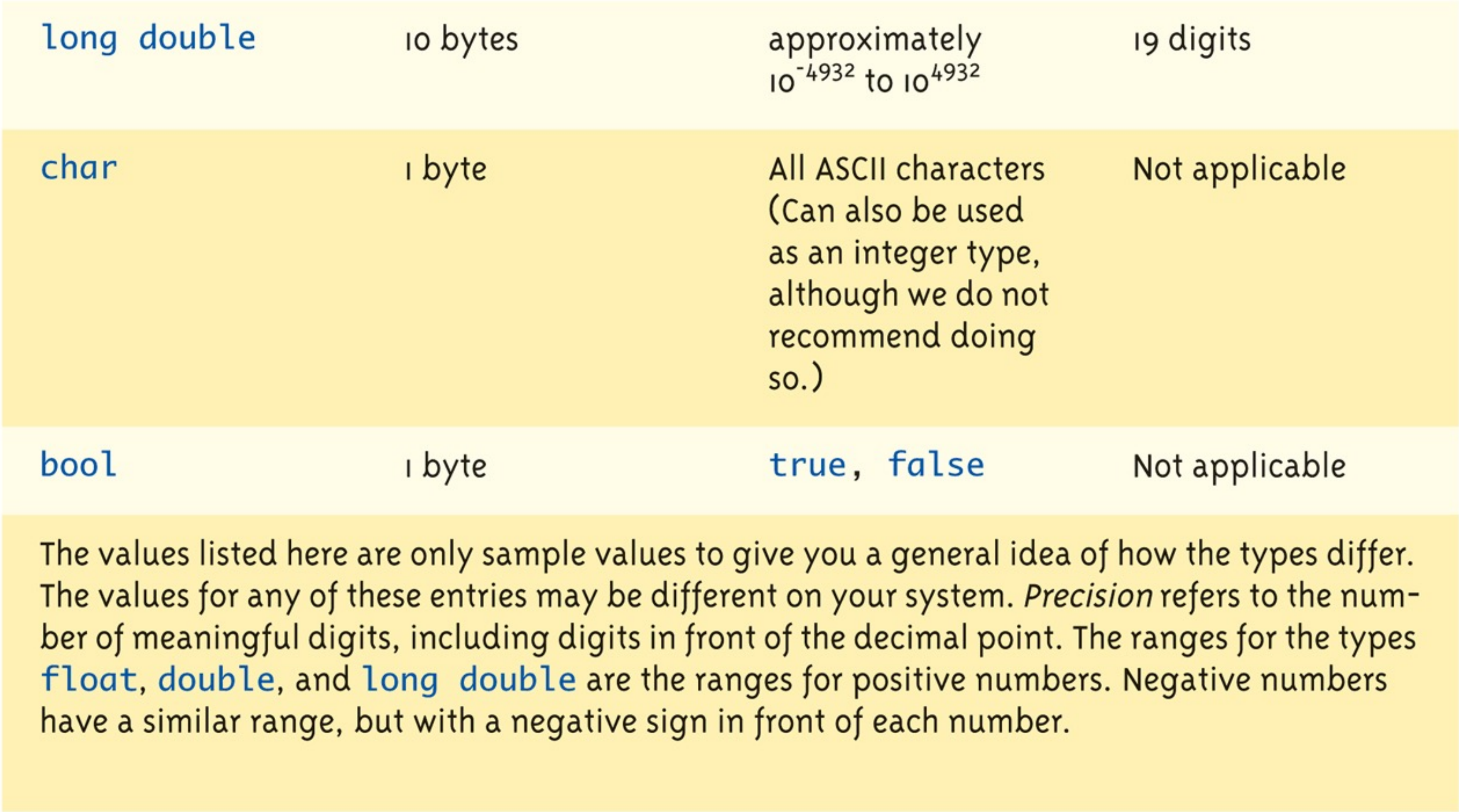


Display 1.2 **Simple Types**

TYPE NAME	MEMORY USED	SIZE RANGE	PRECISION
`short` (also called `short int`)	2 bytes	−32,768 to 32,767	Not applicable
`int`	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`long` (also called `long int`)	4 bytes	−2,147,483,648 to 2,147,483,647	Not applicable
`float`	4 bytes	approximately 10^{-38} to 10^{38}	7 digits
`double`	8 bytes	approximately 10^{-308} to 10^{308}	15 digits

Plain Old Data (POD) Types in C/C++

- Note:
  - bool is not a standard C type, but is in C++

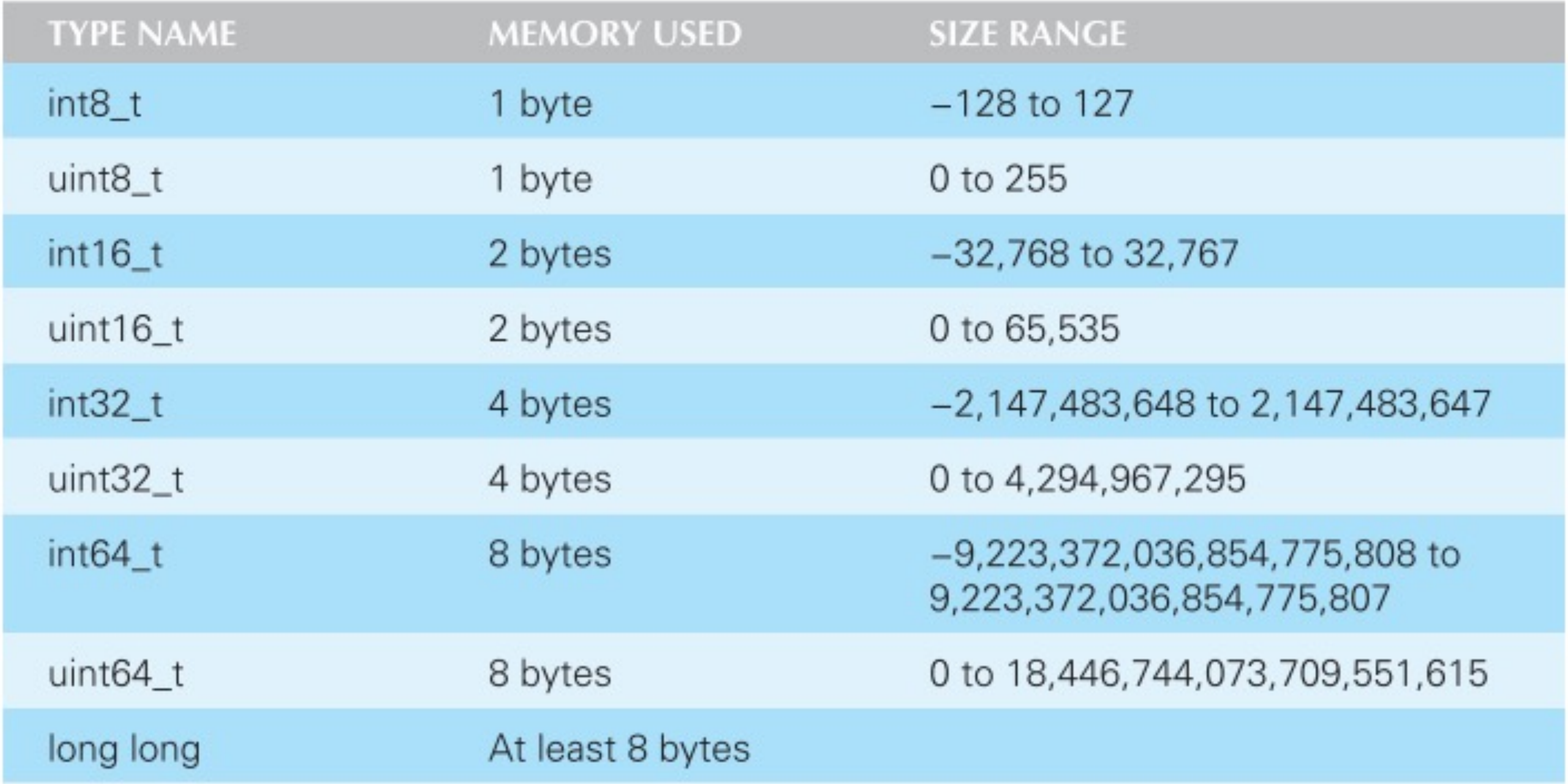


`long double`	10 bytes	approximately 10^{-4932} to 10^{4932}	19 digits
`char`	1 byte	All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)	Not applicable
`bool`	1 byte	`true, false`	Not applicable

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.

New Integral Types (C++11)

- Avoids problem of variable integer sizes for different CPUs
  - Defined in <inttypes.h> or <cinttypes>



TYPE NAME	MEMORY USED	SIZE RANGE
int8_t	1 byte	–128 to 127
uint8_t	1 byte	0 to 255
int16_t	2 bytes	–32,768 to 32,767
uint16_t	2 bytes	0 to 65,535
int32_t	4 bytes	–2,147,483,648 to 2,147,483,647
uint32_t	4 bytes	0 to 4,294,967,295
int64_t	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint64_t	8 bytes	0 to 18,446,744,073,709,551,615
long long	At least 8 bytes	

size_t (since C98)

- an unsigned data type defined by C/C++ standards.
  - typically, it represents the size of types/variables in terms of bytes

```
size_t int_size = sizeof(int);
size_t double_size = sizeof(double);

printf( "%zu %zu\n", int_size, double_size );
```

```
>> 4 8
```

Size_t example

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

int main()
{
  size_t short_size = sizeof(short);
  size_t int_size = sizeof(int);
  size_t long_size = sizeof(long);
  size_t float_size = sizeof(float);
  size_t double_size = sizeof(double);
  size_t long_double_size = sizeof(long double);
  size_t char_size = sizeof(char);
  size_t bool_size = sizeof(bool);
  size_t int16_size = sizeof(int16_t);
  size_t uint32_size = sizeof(uint32_t);
  size_t int64_size = sizeof(int64_t);
  size_t longlong_size = sizeof(long long);

  cout << short_size << "(shrt) " << int_size << "(int) " << long_size << "(long) "
<< float_size << "(float) " << double_size << "(double) " << long_double_size << "(long
double) " << char_size << "(char) " << bool_size << "(bool) " << int16_size << "(int16) "
<< uint32_size << "(unit32) " << int64_size << "(int64) " << longlong_size << "(long long) "
<< endl;
  return 0;
}
```

Raw String Literals (C++11)

- Newly introduced with C++11
- Avoids escape sequences by literally interpreting everything in parentheses

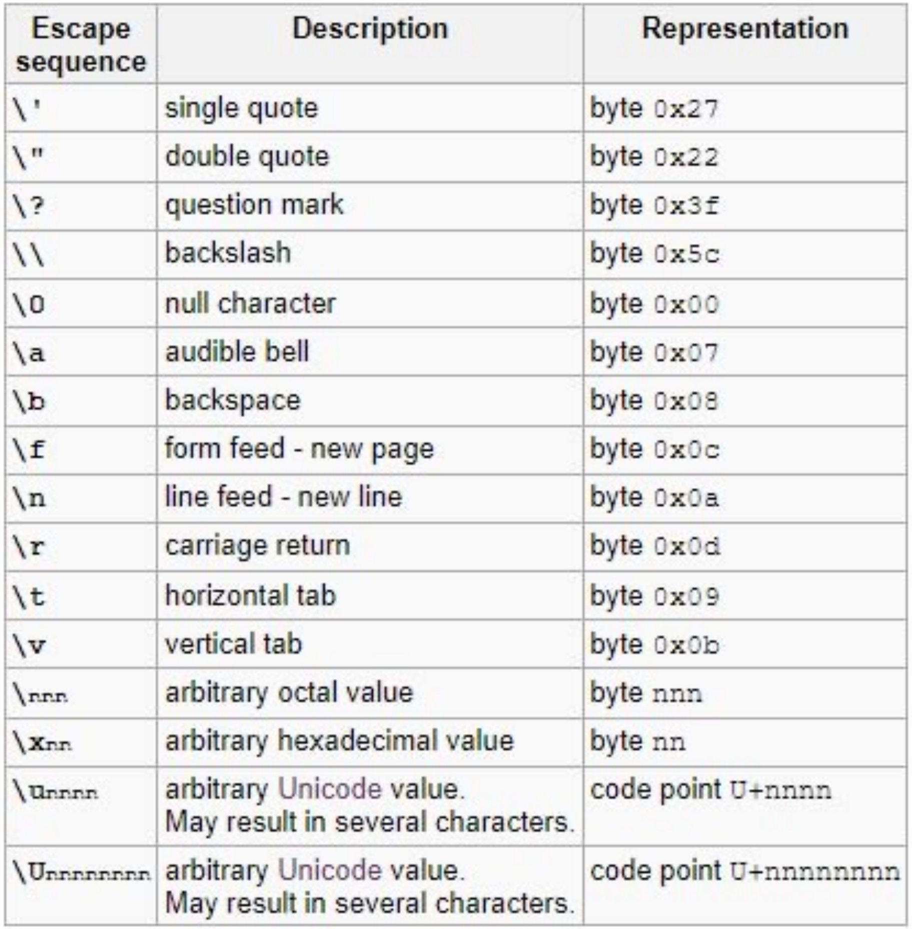
```
string s = R"(\t\\t\n)";
```

  - The variable s is set to the exact string "`\t\\t\n`"
  - Without "R", s is interpreted as tap, backslash, 't', newline.

- Useful for filenames with \ in the file path

Raw String Literals (C++11)

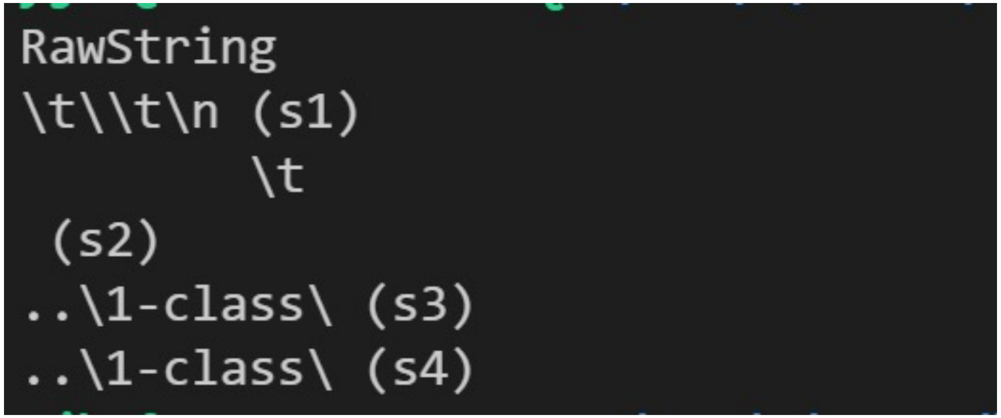
- Escape sequences



Escape sequence	Description	Representation
\'	single quote	byte 0x27
\"	double quote	byte 0x22
\?	question mark	byte 0x3f
\\	backslash	byte 0x5c
\0	null character	byte 0x00
\a	audible bell	byte 0x07
\b	backspace	byte 0x08
\f	form feed - new page	byte 0x0c
\n	line feed - new line	byte 0x0a
\r	carriage return	byte 0x0d
\t	horizontal tab	byte 0x09
\v	vertical tab	byte 0x0b
\nnn	arbitrary octal value	byte nnn
\xnn	arbitrary hexadecimal value	byte nn
\unnnn	arbitrary Unicode value. May result in several characters.	code point U+nnnn
\Unnnnnnnn	arbitrary Unicode value. May result in several characters.	code point U+nnnnnnnn

Raw String example

```
#include <iostream>

using namespace std;

int main()
{
  string s1 = R"(\t\\t\n)";
  cout << s1 << " (s1)" << endl;

  string s2 = "\t\\t\n";
  cout << s2 << " (s2)" <<endl;

  string s3 = R"(..\1-class\)";
  cout << s3 << " (s3)" <<endl;

  string s4 = "..\\1-class\\";
  cout << s4 << " (s4)" <<endl;

  return 0;
}
```



```
RawString
\t\\t\n (s1)
        \t
 (s2)
..\1-class\ (s3)
..\1-class\ (s4)
```

VARIABLE DECLARATION AND TYPE DEDUCTION

Variable Declaration Anywhere

- In C, you needed to pre-declare all the variables before you use in your functions. (C99 allows it.)
  - C++ relaxes this constraint significantly.
  - You can declare variables (nearly) anywhere, as long as syntax allows.
- Example: for loop
  - C style

```
int k;
...
for( k=0; k < 10; k++ ) printf( “%d\n”, k );
```

  - C++ style

```
for( int k=0; k < 10; k++ ) printf( “%d\n”, k );
```

Automatic Type Deduction (C++11)

- new ‘auto’ keyword
  - auto in C (meaning local variable) deprecated
  - “**Auto type inference**”: Deduces the type of the variable based on the expression on the right side of the assignment statement

```
auto x = expression;
```

  - More useful later when we have verbose types
    - e.g., iterators in STL (standard template library)

```
std::unordererd_map<int,std::string>::iterator it = m.begin();
auto it2 = m.begin(); // type deduced automatically
```

Automatic Type Deduction (C++11)

- ‘decltype’
  - automatic type deduction by the existing variable or expression
  - Determines the type of the expression. In the example below, x*3.5 is a double so y is declared as a double.

```
double x=1.0;
decltype(x*3.5) y=2.0;
```

auto example

```
#include <iostream>

using namespace std;

int main()
{
  auto x = 10;
  auto y = "Ten";
  cout << x << " is " <<  y << endl;
}
```

COMMENTS AND STYLES

Comments

- Two methods:

```
// Two slashes indicate entire line is to be ignored
/* Delimiters indicates everything between is ignored */
```

  - Both methods commonly used
  - Note: /// is the same as //, but often used as a meta information indicator.
    - the comment following /// is used for automatic document generation (e.g., doxygen).

Program Style

- Remember the following basic rule:
  - Make programs easy to read, modify, and maintain!

- Naming conventions of identifiers (Book's author)
  - constants: ALL_CAPS
  - variables: lowerToUpper or under_bar_var
  - Most important: make MEANINGFUL NAMES!
    - In case the variable name is meaningful, you do not have to add much comments to explain what the variables are for.

NAMESPACES AND LIBRARIES

Namespaces

- Namespaces defined:
  - Collection of name definitions
  - For now: interested in namespace "std"
  - Has all standard library definitions we need
- Examples:

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

  - Includes entire standard library of name definitions.

```
#include <iostream>
using std::cin;
using std::cout;
```

  - Can specify just the objects we want.

Namespace example

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

void printAll() {
    cout << "printAll 함수" << endl;
}

void printAll() {
    cout << "printAll 함수" << endl;
}

int main(void)
{
    //printAll 함수 호출
    printAll();

    return 0;
}
```

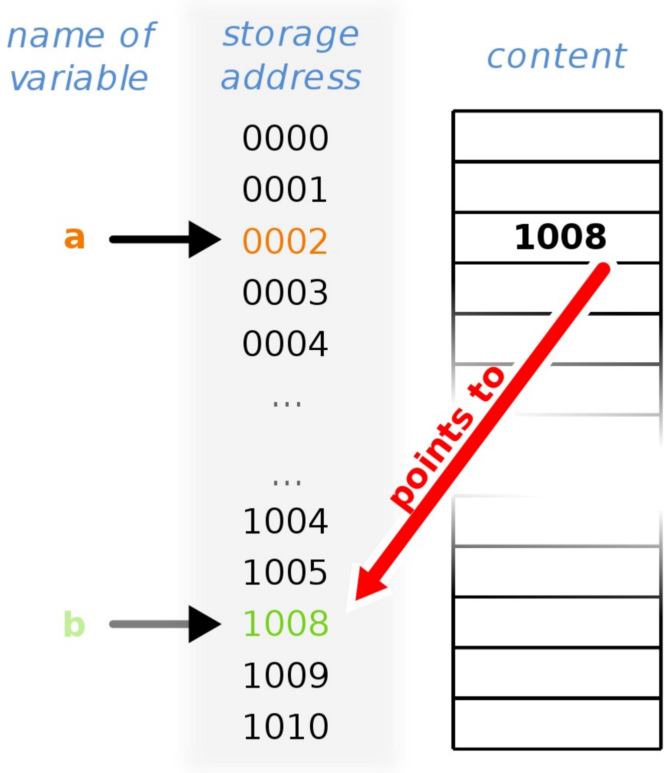
Libraries

- C++ Standard Libraries
- #include <Library_Name>
  - Directive to "add" contents of library file to your program
  - Called "preprocessor directive"
    - Executes before compiler, and simply "copies“ library file into your program file
- C++ has many libraries
  - Input/output, math, strings, etc.

POINTERS AND REFERENCE

Review on Pointers

- Pointer:
  - a data type whose value refers to the address of another value stored elsewhere in the memory.
  - Easier definition: a variable which can store the address of a variable



■ pointer example

DEMO

```
#include <iostream>

using namespace std;

int main()
{
    int* p = NULL;
    int i = 10;
    p = &i;

    cout << ”Storage Address of variable i :" << &i << endl;
    cout << ”Value of pointer p :" << p << endl;
    cout << ”reference value of pointer p :" << *p << endl;

    return 0;
}
```

Review on Pointers

- C-style pointers are still intensively used:

```
int robert, william;
int* bob = &robert,
int *bill = &william;
```

- Pointers are still used for call-by-reference in functions.

Call-by-Value example

DEMO

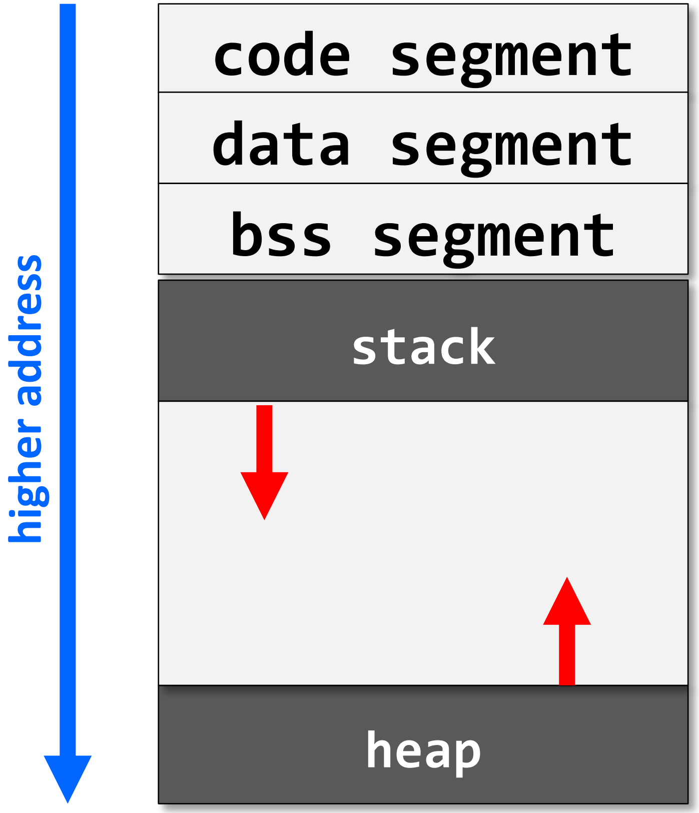
```
#include <iostream>
using namespace std;

void swap(int a, int b);

int main()
{
    int val1 = 10;
    int val2 = 20;
    swap(val1, val2);

    cout<<"val1 : "<<val1<<endl;
    cout<<"val2 : "<<val2<<endl;

    return 0;
}

void swap (int a, int b)
{
    int temp = a;
    a = b;
    b = temp;

    cout<<"a : "<<a<<endl;
    cout<<"b : "<<b<<endl;
}
```

Call-by-Reference example

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

void swap(int *a, int *b);

int main()
{
    int val1 = 10;
    int val2 = 20;

    cout<< "Before the swap function" <<endl;
    cout<<"val1 : "<<val1<<endl;
    cout<<"val2 : "<<val2<<endl;

    swap(&val1, &val2);

    cout<<endl<<"After Swap function " <<endl;
    cout<<"val1 : "<<val1<<endl;
    cout<<"val2 : "<<val2<<endl;

    return 0;
}

void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Review on Pointers: Memory Structure

- A computer program in memory consists of segments



- program instructions
- global/static variables with initialization
- global/static variables (uninitialized)
- local variables in functions
  grows downwards
- dynamically-allocated memory
  grows upwards

Review on Pointers

- Why are pointers so confusing?
  - Reason: usage of ‘*’ is different between declaration and dereference.
  - Declaration:

```
int *p1, *p2; // * is used for declaration
```

  - Dereference

```
int  q = *p1; // * is used for dereference
```

- Good practice to distinguish them
  - Actually, pointer is a different type.
  - So, use int* instead of int * (i.e., remove space in declaration)

```
int* p1;  // int* considered a type, avoiding confusion
int* p2;
```

Reference

- Reference defined:
  - conceptually similar to pointer, but is simpler
    - We can use it, as if it has the same type as the source has.
  - Specified by ampersand (&) after type
  - Name of a storage location or alias to a variable
  - Must be a valid reference! no null/invalid reference exist.
    - Pointer can have a null pointer, but is not for references.

  - Wrong Usages.
    - Int& b=NULL;
    - Int& b;
    - Int& b=10

Reference

- Example of a standalone reference:

```
int robert, william;
int& bob = robert, &bill=william;
```

  - Changes made to bob will affect robert
  - Multiple reference declaration is similar to those of pointers.
    - e.g., int &r1 = vlaue1, &r2 = vlaue2;

reference example

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

int main() {
  int value1 = 5, value2 = 9; // normal integer
  int& ref1 = value1, &ref2 = value2; // reference to variable value

  value1 = 6; // value1 is now 6
  ref1 = 7; // value is now 7

  cout << value1; // prints 7
  ++ref;
  cout << value1; // prints 8

  cout << &value2; // prints 0012FF7C
  cout << &ref2; // prints 0012FF7C
  return 0;
}
```