C++ Programming:

From C to C++

2019년도 2학기

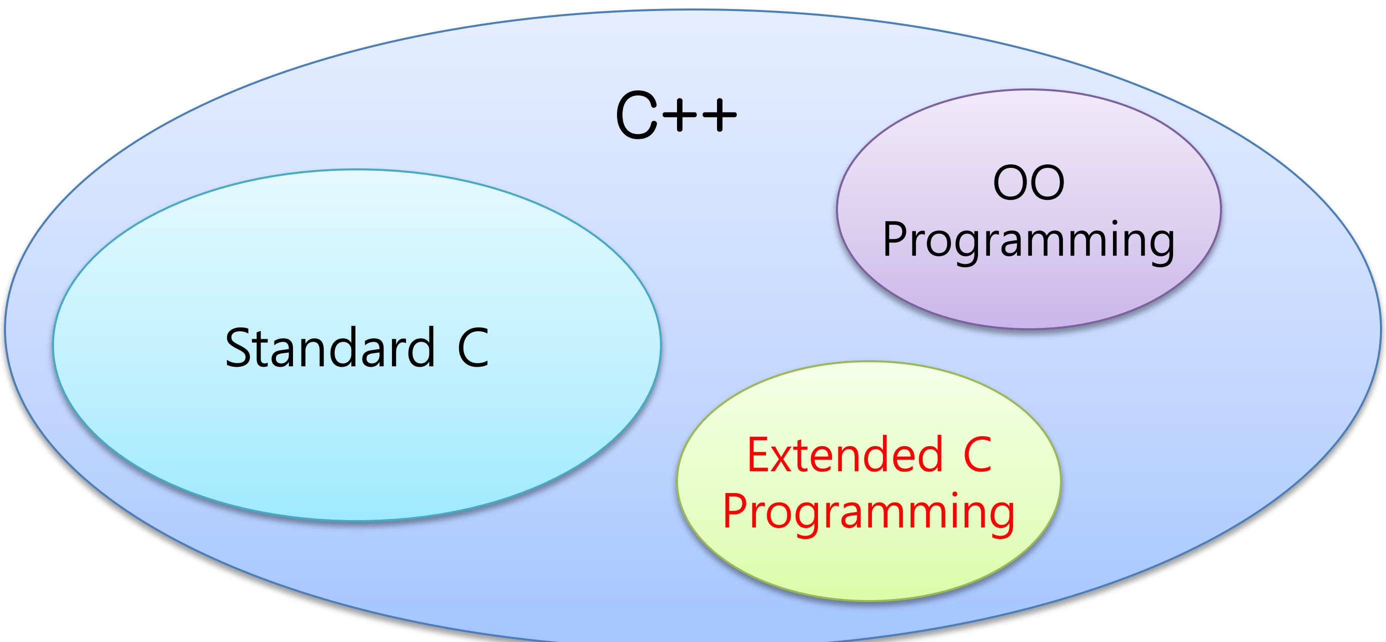
Instructor: Young-guk Ha
Dept. of Computer Science & Engineering
KONKUK UNIVERSITY

Contents

- Discusses some features of C++ extended from ANSI standard C language including:
  - New comment style
  - Namespaces
  - C++ standard input/output library and files
  - New casts, usage of constants, and types (**bool** and **string**)
  - New features in variable and structure definitions
  - New features in function definition
  - Memory allocation operators (**new** and **delete**)
  - Exception handling

C And C++

- Recall that C++ contains standard C as a subset
  - C programs are also C++ programs (not vice versa)
  - Programmers can rewrite fragments of C programs with their C++ counterparts



New Comment Style

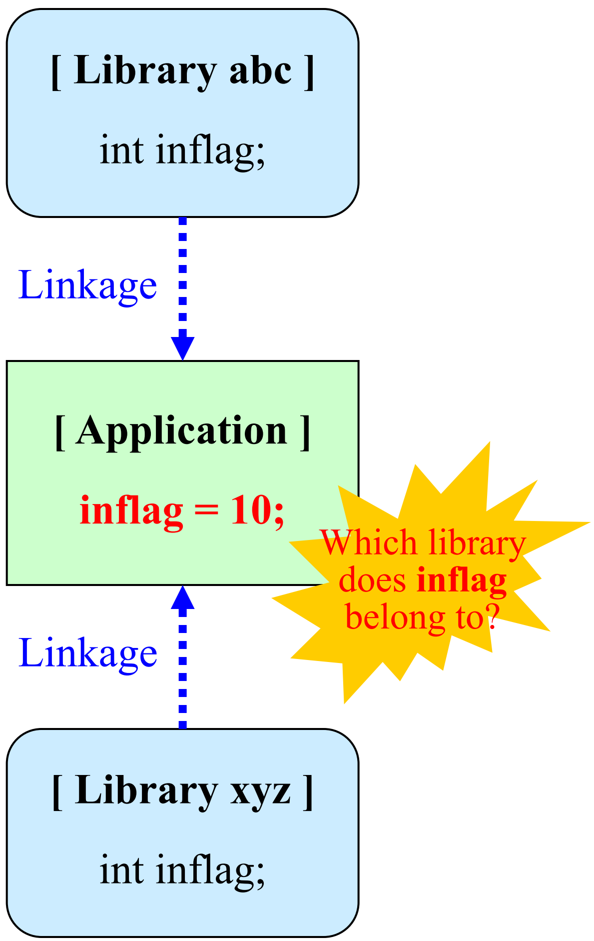
- In addition to the C-style comments ("/* … */"), C++ comments may begin with "//" and extend to the end of the lines

```
// This program outputs the message
//
//        C++: one small step for the program,
//        one giant leap for the programmer
//
// to the screen
#include <iostream>
using namespace std;

int main() {
        cout << "C++: one small step for the program,\n"
             << "one giant leap for the programmer\n";

        return 0;
}
```

Namespaces

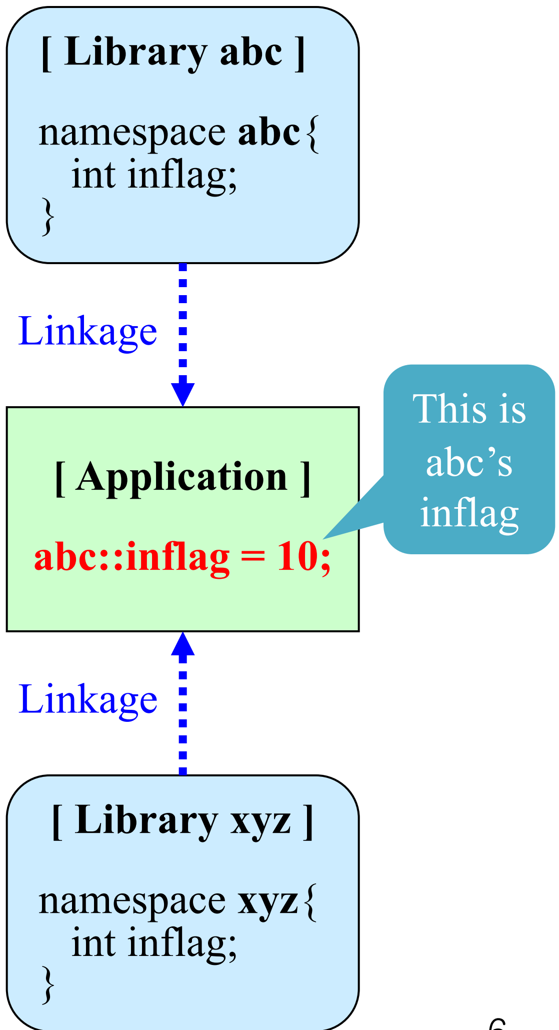
- Name conflict problem
  - Commonly occurs when an application tries to use more than two libraries that have identifiers with the same name
  - A potential problem in large applications involving a number of programmers (must use unique names)

- C++ provides a method to prevent the name conflict: ***namespaces***
  - By disambiguating a name using a unique *prefix* followed by the *scope resolution operator* "**::**"
  - Identifiers in the standard C++ libraries are covered by namespace "**std**" by default (e.g., **cout** in the **iostream** library is actually identified as **std::cout**)



Namespaces (Cont.)

- Declaration of namespaces
  - Enclosing identifiers with its namespace
    - A namespace begins with keyword **`namespace`** followed by a name that identifies the namespace (used as prefix)
  - Whatever can be declared or defined outside a namespace can be declared or defined inside it
    - Functions, variables, types, …
- 3 methods to use namespaces
  - Scope resolution operator
  - Using declaration
  - Using directive



Namespace Examples

```
namespace abc {
    int inflag;
    void g(int);
     ...
}
```

Library *abc*

```
namespace xyz {
    int inflag;
     ...
}
```

Library *xyz*

C++
Applications

```
int main(void)
{
    abc::inflag = 3;
    xyz::inflag = -823;
     ...
}
```

(a) Scope resolution OP

```
using abc::inflag;

int main(void)
{
    inflag = 3;
    abc::g(8);
    xyz::inflag = -823;
     ...
}
```

(b) Using declaration

```
using namespace abc;

int main(void)
{
    inflag = 3;
    g(8);
    xyz::inflag = -823;
     ...
}
```

(c) Using directive

C++ iostream Library Examples

- What problem does this program have and how to fix it?

```
#include <iostream>

int main(void) {
    ...
    cout << "Test";
    ...
}
```

- Valid C++ programs

```
#include <iostream>

int main(void) {
    ...
    std::cout << "Test";
    ...
}
```

(a) Scope resolution OP

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

int main(void) {
    ...
    cout << "Test";
    ...
}
```

(b) Using declaration

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

int main(void) {
    ...
    cout << "Test";
    ...
}
```

(c) Using directive

Standard Headers

- Standard headers in the latest C++ no longer end with ".h"
  - Thus, example code (b) may cause an error in some strict C++ compilers

- For the compatibility's sake, Visual C++ (6.0 or above ver.) allows us to use ".h" headers as well
  - Example (a) and (b) are both valid in VC++

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

int main(void) {
    ...
    cout << "Test";
    ...
}
```

(a)

```
#include <iostream.h>

int main(void) {
    ...
    cout << "Test";
    ...
}
```

(b)

More Examples on "Using Directive" and "Using Declaration"

```
namespace X {
   int i, j, k;
}

int k;      // global variable k

// use of "using directive"
void f1()
{
   using namespace X;
   i++;
   j++;
   k++;    // Error,
           //  X::k or global k?

   X::k++;
   ::k++;     // global k
}
```

```
// use of "using declaration"
void f2()
{
   int i = 0;  // local i

   using X::i;
   i++;   // Error,
          //  X::i or local i?

   using X::j;
   using X::k;
   j++;
   k++;   // Not an error but …
          //  X::k hides global k
          //  (X::k has priority)
}
```

Nested Namespaces

- Namespaces can be nested and identifiers within namespaces basically restricted by a *block scope*
  - Functions can access identifiers declared within the same namespace or outside of the namespace

```
namespace X {
   void g();
    ...
   namespace Y {
       void f();    // X::Y::f()
       void ff();   // X::Y::ff()
    ...
   }
}

void X::Y::ff()
{
   f();   // within the same NS
   g();   // outside Y
   h();   // outside Y (global)
}
```

```
void X::g()
{
   f();    // Error: not accessible
           //        (not within X)
   Y::f();  // OK with prefix
}

void h()
{
   f();     // Error: no global f()
   Y::f(); // Error: Y is not
           //        global
   X::f(); // Error: f() is not
           //        within X
   X::Y::f();  // OK with prefix
}
```

Openness of Namespaces

```
namespace A {
    int f();
}

    ...

namespace A {
    int g();
    int h();
}
```

(a) Separate namespaces

=

```
namespace A {
    int f();
    int g();
    int h();
}
```

(b) Single integrated namespace

Anonymous Namespaces

- A namespace can be used anonymously
  - The ***anonymous namespace*** (a.k.a. unnamed namespaces) replaces keyword **`static`** of C
- One usage of **`static`** in C language is to restrict functions or global variables to a file scope (called *internal linkage*)
  - This allows multiple library files to have identifiers with the same name
  - However, in C++, this usage of **`static`** is deprecated in favor of anonymous namespaces

```c
// Library C
static void f()
```

```c
// Library B
static void f()
```

```c
// Library A
static void f()
{
   ...
}
```

C program libraries

```cpp
namespace {
    void f()
```

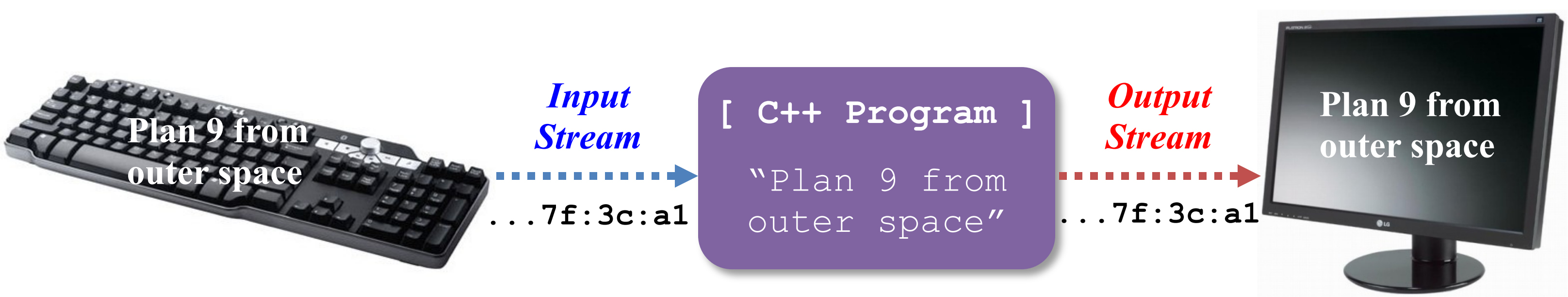
```cpp
namespace {
    void f()
```

```cpp
namespace {
    void f()
    {
       ...
    }
}
```

C++ program libraries

Basic I/O in C++

- C++ provides an alternative to I/O library of C
  - Easier-to-use, extensible, and more flexible
  - This section introduces basic I/O functionalities of C++
    - The details on C++ I/O will be covered later
- Stream-based I/O model
  - Input to a C++ program is treated as a ***stream*** of consecutive bytes from an input device (e.g., keyboard, disk, scanner, …)
  - Output from a C++ program is also treated as a byte stream to an output device (e.g., video display, disk, printer, …)



C++ I/O Libraries

- Standard I/O variables
  - Declared in the "**iostream**" header
    - **cin** for console input
    - **cout** for console output
    - **cerr** for console error
- I/O manipulators (for I/O stream formatting)
  - Formatting numbers: dec, hex, oct, showpoint, noshowpoint
  - Formatting floating-point numbers: setprecision, fixed, scientific
  - General I/O: skipws, noskipws, setw, setfill, flush, endl, …
- More I/O classes and methods
  - Will be covered later

Standard I/O Variables

- Standard I/O variables of C++

	`cin`	`cout`	`cerr`
Description	console input	console output	console error
Default src. or dest.	keyboard	video display	video display
Buffered	O	O	X

- C++ I/O variables vs. Std. I/O descriptors of C
  - **cin** ≒ **stdin**, **cout** ≒ **stdout**, **cerr** ≒ **stderr**
  - More powerful and flexible than I/O descriptors
    - Actually, C++ I/O variables are *objects of I/O stream classes* (including various I/O methods)

C++ I/O Operators

- Standard I/O variables are used with the I/O operators
  - Input operator “**>>**”
    - E.g., **`cin >> x;`** // reads a value from the keyboard and store the value into ***x***
  - Output operator “**<<**”
    - E.g., **`cout << x;`** // writes the value of ***x*** to the display

- I/O operators
  - Left-associated
    - Evaluated from left to right
    - E.g., **`cout << x << y;`** // writes ***x*** first and then ***y*** to the display
  - Automatically recognize type of the data
    - No format string required (cf., <stdio.h> function **printf** or **scanf**)
    - E.g.,

```
int x;
cin >> x;      // == scanf("%d", &x)
cout << x;     // == printf("%d", x)
```

Standard I/O Examples

I/O operators are left-associated, i.e., evaluated *from left to right*

```
#include <iostream>
usig namespace std;
void main()
{
    int id;
    float av;
    cout << "Enter the id and the average: ";
    cin >> id >> av ;
    cout << "Id = " << id << '\n'
         << "Average = " << av << '\n';
}
```

```
#include <iostream>
using namespace std;
void main() {
    int val, sum = 0;
    cout << "Enter next number: ";
    while (cin >> val) {
          sum += val;
          cout << "Enter next number: ";
    }
    cout << "Sum of all values: " << sum << '\n';
}
```

Statement "**cin >> val**" is evaluated as **true**, if an integer value is read correctly, otherwise **false**

I/O Manipulators

- I/O manipulators format inputs and outputs
  - Permanently changes the state of I/O stream to which it is applied
    - ❖ Except **`setw`**: effect of **`setw`** lasts for the next I/O op. only (effects once)
  - To use manipulators without arguments, the header "**`iostream`**" must be included
  - Manipulators with arguments requires the header "**`iomanip`**"

Manipulator	*Effect*
`dec`	Input or output in decimal
`endl`	Write newline and flush output stream (== '**`\n`**')
`fixed`	Use fixed notation for floating-point numbers: **`d.ddd`**
`flush`	Flush output stream
`hex`	Input or output in hexadecimal
`left`	Left-justify
`oct`	Input or output in octal
`right`	Right-justify
`scientific`	Use scientific notation for floating-point numbers: **`d.dddEdd`**
`setfill( c )`	Make **`c`** the fill character
`setprecision( n )`	Set floating-point precision to **`n`**
`setw( n )`	Set field width to **`n`** (and right-justify)
`showpoint`	Always print decimal point and trailing zeros
`noshowpoint`	Don't print trailing zeros. Drop decimal point, if possible.
`showpos`	Use **`+`** with nonnegative numbers
`noshowpos`	Don't use **`+`** with nonnegative numbers
`skipws`	Skip white space before input
`noskipws`	Don't skip white space before input
`ws`	Remove white space

Using I/O Manipulators

E.g. 1: To print integers in various formats

```
int i = 91; // output is in decimal by default
cout << "i = " << i << " (dec)" << endl;
cout << "i = " << oct << i << " (oct)" << endl;
cout << "i = " << hex << i << " (hex)" << endl;
cout << "i = " << i << " (hex)" << endl;
cout << "i = " << dec << i << " (dec)" << endl;
```

```
i = 91 (dec)
i = 133 (oct)
i = 5b (hex)
i = 5b (hex)
i = 91 (dec)
```

E.g. 2-1: To format outputs with right-justification in a field of width 6

```
for (i = 1; i <= 1000; i *= 10)
        cout << setw(6) << i << '\n';
        // == format string "%6d" of printf
```

```
     1
    10
   100
  1000
```

E.g. 2-2: **setw** lasts only for the next operation, not permanently

```
// takes setw out of the for loop
cout << setw(6);
for (i = 1; i <= 1000; i *= 10)
        cout << i << '\n';
```

```
     1
10
100
1000
```

Using I/O Manipulators (Cont.)

E.g. 3: To format outputs with left and right-justifications

```
int a = 5, b = 43, c = 104;
cout << left  << setw(10) << "Karen"
     << right << setw(6)  << a << '\n';
cout << left  << setw(10) << "Ben"
     << right << setw(6)  << b << '\n';
cout << left  << setw(10) << "Patricia"
     << right << setw(6)  << c << '\n';
```

```
12345678901234567890
Karen          5
Ben           43
Patricia     104
```

E.g. 4: To fill extra columns with the given character

```
cout << setfill('*');
for (i = 1; i <= 1000; i *= 10)
        cout << setw(6) << i << '\n';
```

```
12345678901234567890
*****1
****10
***100
**1000
```

E.g. 5: To specify the number of precision digits for floating-point numbers

```
float a = 1.05, b = 10.15, c = 200.87;
cout << setfill('*') << setprecision(2);
cout << setw(10) << a << '\n';
cout << setw(10) << b << '\n';
cout << setw(10) << c << '\n';
```

```
12345678901234567890
******1.05
*****10.15
****200.87
```

Using I/O Manipulators (Cont.)

E.g. 6-1: To print floating-point numbers without a decimal point and trailing 0s

```
float a = 5, b = 43.3, c = 10304.31;
cout << setw(8) << a << '\n';
cout << setw(8) << b << '\n';
cout << setw(8) << c << '\n';
```

```
12345678901234567890
       5
    43.3
1.03e+04
```

E.g. 6-2: To print floating-point numbers in fixed format with a decimal point and trailing 0s

```
float a = 5, b = 43.3, c = 10304.31;
cout << showpoint << fixed << setprecision(2);
cout << setw(8) << a << '\n';
cout << setw(8) << b << '\n';
cout << setw(8) << c << '\n';
```

```
12345678901234567890
    5.00
   43.30
10304.31
```

E.g. 7: To allow white spaces as inputs (do not skip ws)

```
#include <iostream>
using namespace std;
void main() {
          char c;
          cin >> noskipws;
          while (cin >> c) cout << c;
}
```

Mixing C & C++ I/O

- Using both C and C++ I/O libraries in the same program may cause problems
  - Because inputs and outputs of these two libraries are not automatically synchronized
    - E.g., using **printf** and **cout** in one program at the same time
  - To resolve this problem, **std::ios::sync_with_stdio(true)** need to be invoked before mixing I/O operations

```
#include <iostream>
#include <cstdio>      // aka <stdio.h>
...
ios::sync_with_stdio(true);

cout << "value =";
printf("%d\n", i);
...
```

Simple File I/O

- It is possible to read from and write to disk files in the same way as using **`cin`** and **`cout`**
  - At first, "**`fstream`**" header needs to be included
  - Then, we can just replace …
    - **`cin`** with an **`ifstream`** object associated with the input file, or
    - **`cout`** with an **`ofstream`** object associated with the output file
  - The I/O operators "**>>**" and "**<<**" are used in just the same way as they are used with **`cin`** and **`cout`**

File I/O Example

```
// This program repeatedly reads an income from
// the file income.in until end-of file.  Income
// under 6000 greenbacks is taxed at 30 percent,
// and income greater than or equal to 6000
// greenbacks is taxed at 60 percent.  After
// reading each income, the program prints the
// income and tax.
#include <fstream>
using namespace std;
const int cutoff = 6000;
const float rate1 = 0.3;
const float rate2 = 0.6;
int main() {
    ifstream infile;
    ofstream outfile;
    int income, tax;
    infile.open( "income.in" );
    outfile.open( "tax.out" );
    while ( infile >> income ) {
        if ( income < cutoff )
            tax = rate1 * income;
        else
            tax = rate2 * income;
        outfile << "Income = " << income
                << " greenbacks\n"
                << "Tax = " << tax
                << " greenbacks\n";
    }
    infile.close();
    outfile.close();
    return 0;
}
```

1) Include the **fstream** header and use <u>using directive</u> for the **std** namespace

2) Generate objects of **ifstream** and **ofstream** classes

3) Open input and output files

```
// The following code to test whether
// the file is open can be added here
//  (infile is converted to true if the
//   file is successfully open, false
//   otherwise)
if (!infile) {
  cerr << "Unable to open income.in\n";
  exit(0);
}
```

4-1) Read from the input file stream

4-2) Write to the output file stream

5) Close input and output files

Type Casts in C++

- Static cast (**`static_cast`**)
  - Same as common type cast in C language
    - E.g., converting type **`int`** to **`float`**

```
int   int_val   = 123;

/* Common type cast in C language syntax */
float float_val = (float)int_val;

// C++ static_cast
float float_val = static_cast<float>(int_val);
```

- Special-purpose casts newly suppoted in C++
  - Constant cast (**`const_cast`**)
  - Reinterpret cast (**`reinterpret_cast`**)
  - Dynamic cast (**`dynamic_cast`**)
    - Used for casting across or within inheritance hierarchies (will discus later)

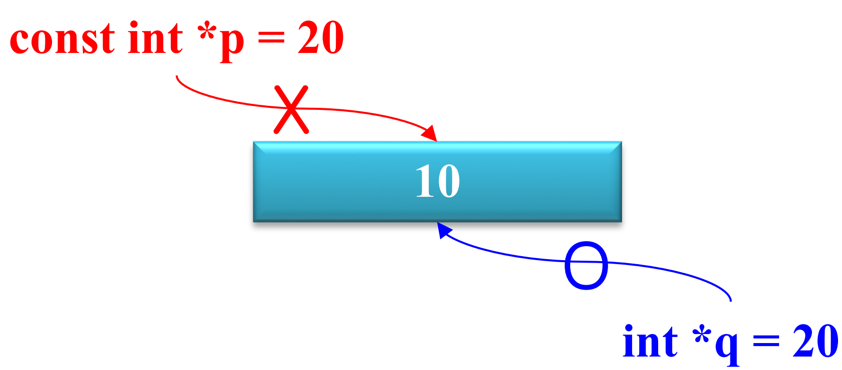
Constant Cast

- Used to cast away constantness of a **`const`** pointer to an object or variable

Compile error: a **`const`** pointer can not be placed on the left-hand side of “**=**”

```
const int* p = new int(10);

*p = 20;

int* q = const_cast<int*>(p);
*q = 20;
```

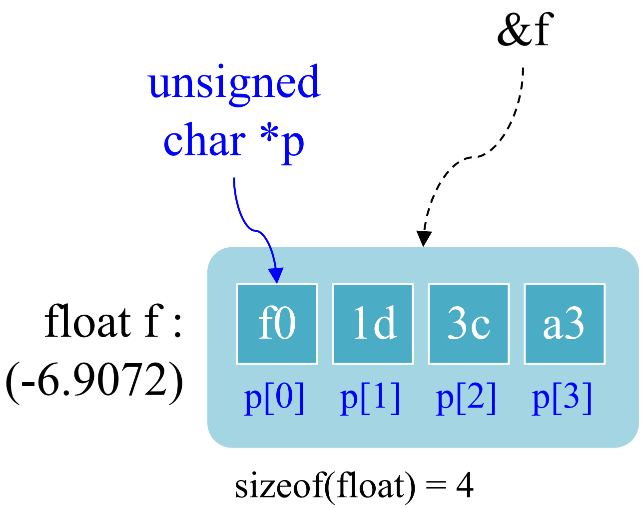
`const_cast` is used to cast away constantness of **`p`**, a **`const`** pointer to an **`int`**



Reinterpret Cast

- Used to convert
  - From a pointer of one type to a pointer of another type
  - From a pointer type to an integer type
  - From an integer type to a pointer type

❖ *Note that the effect is system dependent (e.g., endianness), so* ***reinterpret_cast*** *should be used with caution*




```
int i;
float f = -6.9072;      // sizeof(float) = 4 bytes
unsigned char* p = reinterpret_cast<unsigned char*>(&f);

cout << hex;            // prints each byte of f in hex
for (i = 0; i < sizeof(float); i++)
       cout << static_cast<int>(p[i]) << "\n";
```

Usage of Constants

- Unlike C, a **`const`** variable can be used anywhere a constant can appear
  - E.g. 1, used as an array size

```
const int s = 100;
float f[s];
```

  - E.g. 2, used as an expression in case label

```
switch (a) {
  case s:        ... ;
  case s+1:      ... ;
  // ...
}
```

Changes in Declarations and Definitions

- Enumerated types
  - Variables of an enumerated type can be declared without the **`enum`** keyword
    - E.g., enumerated type in C

```
enum marital_status   { single, married };        // definition of an enum type
enum marital_status   person1;                    // declaration of an enum variable
```

    - E.g., using enumerated type in C++

```
marital_status   person1;             // declaration w/o keyword enum
```

  - **`enum`** can be used anonymously to define constants
    - E.g., declarations of constants with anonymous **`enum`**

```
enum { MinSize = 0, MaxSize = 1000 };
                              // same as "const int MinSize=0, MaxSize=1000;"
int minVal = MinSize, arr[MaxSize];
```

Changes in Declarations and Definitions (Cont.)

- Variables
  - Declarations of variables may occur anywhere in a block, i.e., no need to appear in the beginning of a block
  - However, they must appear prior to their first usages

```
void reverse_and_print(int a[], size) {
    for (int i = 0; i < size; i++) a[i] = 2 * i;
    int temp;                              // temp is defined prior to its first use
    for (int i = 0; i < size / 2; i++) {   // i is defined in each for loop header
            temp = a[i];                   //     (its scope is body of each for loop)
            a[i] = a[size - 1 - i];
            a[size - 1 - i] = temp;
    }
    for (int i = 0; i < size; i++) cout << a[i] << '\n';
}
```

Changes in Declarations and Definitions (Cont.)

- Structures
  - A minor change is that keyword **`struct`** does not need to be used in structure variable declarations

```
struct Point {
      double x, y;
};
Point p1;     // no "struct", cf., in C, this should be "struct Point p1;"
```

  - A major change is that <u>C++ structures can contain function members</u> as well as data members (cf., class)

```
struct Point {
   double x, y;
   void setVal(double, double);
};
```

=

```
class Point {
  public:
     double x, y;
     void setVal(double, double);
};
```

```
Point p1;
p1.setVal(3.1, -15.2);    // invokation of a member function of p1
```

Boolean Data Type

- Boolean data type "**bool**"
  - In C, nonzero represents **true** and zero represents **false**
  - C++ has a new type **bool** to represent **true** and **false** values
    - All relational, equality and logical operators now return a **bool** type result, not **int**
    - However, integer expressions are still permitted where a **bool** value is expected
      - By default, **true** is treated as 1 and **false** is treated as 0

```
bool flag;
flag = (3 < 5);
cout << flag << "\n";
cout << boolalpha << flag << "\n";    // boolalpha manipulator reads and writes
                                      //     bool values as "true" and "false"
                                      //     (cf., noboolalpha for 1 and 0)
```

```
1
true
```

String Data Type

- C-style strings
  - Null-terminated arrays of characters (**`char`** type)
    - E.g.) char str[80]; strcpy(str, “This is a string copy example.”);

- C++ provides a new type **`string`** as an alternative to the C-style strings
  - Requires to include the header “**`string`**”
  - Actually, type **`string`** is defined as a class
    - Thus, any **`string`** variable is a <u>**`string`** object</u>
    - Class **`string`** provides various methods to handle the values of **`string`** objects
      - To modify the string: `insert, replace, erase, swap`
      - To search the string: `find, rfind, find_first_of, find_first_not_of`
      - Other methods: `length, c_str, substr,` …

String Data Type (Cont.)

- By using the **`string`**, programmers do not have to be concerned about the following jobs, which are delegated to the system
  - Storage allocation and deallocation for strings
  - Handling the annoying null terminators
  - Internal representation or encoding of strings

- E.g., declaration of string variables

```
#include <string>
using namespace std;
string s1;                // s1 is declared w/o an initial value (null string)
string s2 = “Bravo”;      // s2’s initial value is a C-style string “Bravo”
string s3 = s2;           // both s3 and s2 represent “Bravo”
string s4(10, ‘x’);       // s4 represents a string consisting of 10 x’s (i.e., “xxxxxxxxxx”)
```

String Data Type (Cont.)

- Conversion to C-style strings
  - Sometimes C-style string is required
  - The member function **`c_str`** returns a pointer to a **`const`** null-terminated array of **`char`** representing the string
    - E.g., when opening a file, the filename must be a C-style string

```
string filename = "infile.dat";
ifstream infile;
infile.open(filename);                // Illegal
infile.open(filename.c_str());        // OK
```

- String length
  - The member function **`length`** returns the length of the string

```
string s = "Ed Wood";
cout << "Length = " << s.length() << '\n'
// output: Length = 7
```

String Data Type (Cont.)

- Writing strings using operator “<<”

  string s1; // null string
  string s2 = “Bravo”;
  string s3 = s2;
  string s4(10, ‘x’);
  cout << s1 << ‘\n’ << s2 << ‘\n’ << s3 << ‘\n’ << s4;

```
Bravo
Bravo
xxxxxxxxxx
```

- Reading strings using operator “>>”

  string s;
  cout << "Enter a string: ";
  cin >> s; // Skips white spaces: even if we enter “Ed Wood”, ***s*** only represents “Ed”

- Reading a line of string using function **`getline`**

  string buff;
  // **getline** reads chars from the input stream until **EOF is reached** (returns **false**)
  // or a **newline** char is read (**newline** is not stored in ***buff***)
  while (**getline(cin, buff)**) cout << buff << "\n";

String Data Type (Cont.)

- Assignment (copy) using "=" operator

```
string s1, s2;
s1 = “Ray Dennis Steckler”;   // right-hand side can be a string, a C-style string, or a char
s2 = s1;
cout << s1 << ‘n’;
cout << s2 << ‘n’;
```

```
Ray Dennis Steckler
Ray Dennis Steckler
```

- Concatenation using "+" operator

```
string s1 = “Atlas”, s2 = “King”, s3;
s3 = s1 + ‘ ’ + s2;          // an operand of “+” can be a string, a C-style string, or a char
cout << s1 << '\n';
cout << s2 << '\n';
cout << s3 << '\n';
s1 += s2;
cout << s1 << '\n';
cout << s2 << '\n';
```

```
Atlas
King
Atlas King
AtlasKing
King
```

String Data Type (Cont.)

- Modification of strings
  - Deleting a substring from the string

```
string s = "Ray Dennis Steckler";
s.erase(4, 7);
cout << s << '\n';
```

```
Ray Steckler
```

  - Inserting a string at a given position

```
string s1 = "Ray Steckler", s2 = "Dennis ";
s1.insert(4, s2);
cout << s1 << '\n';
```

```
Ray Dennis Steckler
```

  - Replacing a substring with a given string

```
string s1 = "Ray Dennis Steckler", s2 = "Fran";
s1.replace (4, 6, s2);
cout << s1 << '\n';
```

```
Ray Fran Steckler
```

  - Swapping two strings

```
string s1 = "Ray Dennis Steckler", s2 = "Ed Wood";
s1.swap(s2);
cout << s1 << '\n' << s2 << '\n';
```

```
Ed Wood
Ray Dennis Steckler
```

String Data Type (Cont.)

- Referencing a char at a specified index like an array

```
string s = "Nan";
cout << s[1] << '\n';
s[0] = 'J';
cout << s << '\n';
```

```
a
Jan
```

- Extracting a substring

```
string s1 = "Ray Dennis Steckler";
string s2;
s2 = s1.substr(4, 6);
cout << s1 << '\n';
cout << s2 << '\n';
```

```
Ray Dennis Steckler
Dennis
```

String Data Type (Cont.)

- Search a string (1)
  - **s1.find(s2, *ind*), s1.find(s2)**
    - Searches string **s1** for a substring **s2** from the beginning of **s1**
    - If **s2** is found, returns the smallest index ≥ ***ind*** where **s2** begins, if **s2** is not found, "plus infinity"
    - Note that **s1.find(s2)** = s1.find(s2, 0)
  - **s1.rfind(s2, *ind*), s1.rfind(s2)**
    - Searches string **s1** for a substring **s2** from the end of **s1** reversely
    - If **s2** is found, returns the largest index ≤ ***ind*** where **s2** begins, if **s2** is not found, "plus infinity"
    - Note that **s1.rfind(s2)** = s1.rfind(s2, s1.length()-1)

```
string s1 = "Ray Dennis Dennis Steckler";
string s2 = "Dennis";
int f = s1.find(s2);
int r = s1.rfind(s2);
cout << "Found at index: " << f << '\n';
cout << "Found at index: " << r << '\n';
```

```
Found at index: 4
Found at index: 11
```

String Data Type (Cont.)

- Search a string (2)
  - **s1.find_first_of(s2)**
    - Returns the index of the first char in **s1** that is also in **s2**
    - If it fails, "plus infinity"
  - **s1.find_first_not_of(s2)**
    - Returns the index of the first char in **s1** that is ***NOT*** in **s2**
    - If it fails, "plus infinity"

```
string s1 = "Abby";
string s2 = "bx";
cout << s1.find_first_of(s2) << '\n';          // the 1st char in s1 and also in s2 = 'b'
cout << s1.find_first_not_of(s2) << '\n';      // the 1st char in s1 but not in s2 = 'A'
```

```
1
0
```

String Data Type (Cont.)

- Comparisons between strings
  - The comparison operators (“**==**“, “**!=**“, “**<**“, “**<=**“, “**>**“, and “**>=**“) can be used to compare strings
    - On each side of the operators, either string or C-style string can appear
  - The comparisons are based on ***lexicographic order***, i.e., the order in which the strings to be compared appear on the dictionary

```
string s1 = “panorama”;
string s2 = “pants”;

// Comparisons in a lexicographic order
cout << boolalpha << (s1 == s2) << ‘\n’;
cout << boolalpha << (s1 != s2) << ‘\n’;
cout << boolalpha << (s1 > s2) << ‘\n’;
cout << boolalpha << (s1 < s2) << ‘\n’;
```

```
false
true
?
?
```

Extended Features in Functions

- Function declarations
- References
  - Call by reference
  - Return by reference
- Inline functions
- Default arguments
- Function overloading

C++ Function Declarations

- Function prototypes
  - The style of declaring functions and writing function headers in which the data types of the parameters are included in the parenthesis
  - In C++, prototypes are ***REQUIRED***
- Functions with no parameters
  - Can have an empty parameter list
  - No need to specify **`void`**
- Return type of functions
  - ***MUST*** be specified
  - A missing return type does ***NOT*** default to **`int`**

Function Declaration Examples

```c
/*
     C-style function
     declarations
*/

void print(void)
{
   /* ... */
}

main(argc, argv)
int argc;
char* argv[];
{
   /* ... */

   return 0;
}
```

```cpp
//
//   C++-style function
//   declarations
//

void print() // Empty parameter
{
   // ...
}

// Prototyping and a return type
//  are required
int main(int argc, char* argv[])
{
   // ...

   return 0;
}
```

References

- A ***reference*** is an alternative way to access storage
  - Declared with the ampersand (**&**) as a variable
- References operate somewhat like pointers
  - Except that no ***dereferencing*** is required
    - Dereferencing of a pointer is to prefix the pointer with "*"
  - Useful in passing arguments to functions (*call by reference*) and returning values from functions (*return by reference*)

```
int x = 0;

int* pnt = x;   // Declaring a pointer to the variable x
*pnt = 5;       // dereferencing required: pnt → *pnt

int& ref = x;   // Declaring a reference to the variable x
ref = 3;        // Accessing x with the reference:
                //             no dereferencing is required
```

Call by Reference

- With "***call by reference***", parameters of a function refer to the actual arguments passed to the function
  - By default, copies of the arguments are passed as "***call by value***"
  - So far in C, we have implemented "*call by reference*" with pointers
  - From now in C++, we can implement "*call by reference*" with references (no dereferencing required)

```
void main() {
    int i = 7, j = -3;
    swap(i, j);
    cout << "i = " << i << '\n'
         << "j = " << j << '\n';
}

// Call by value version of swap
// : Why does this func. fail?
void swap(int a, int b) {
    int t;
    t = a; a = b; b = t;
}
```

```
// Pointer version of swap
void swap(int* a, int* b) {
    int t;
    t = *a; *a = *b; *b = t;
}

// Reference version of swap
void swap(int& a, int& b) {
    int t;
    t = a; a = b; b = t;
    // No dereferencing
}
```

Call by Reference (Cont.)

- Some kinds of arguments (objects) ***MUST*** be passed with call by reference
  - E.g., an I/O stream object must remain changed to keep track of the details such as
    - Formatting information changed by I/O manipulators
    - Current position of the I/O stream

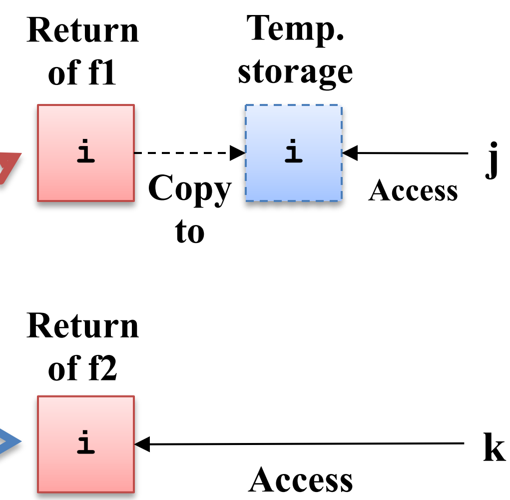
```
void print_row(ifstream& in, ofstream& out, int n)
{
    char c;
    out << right;          // formatting info. changes
    for (int i = 0; i < n; i++) {
        in >> c;           // pos. of the ifstream changes
        out << c;          // pos. of the ofstream changes
    }
    out << '\n';           // pos. of the ofstream changes
}
```

Return by Reference

- Return by value
  - By default, a return value of a function is copied into temporary storage in C and C++
  - And then the invoking function can access the copied return value
- ***Return by reference***
  - The return value is not copied, rather the actual storage is made available to the invoking function

```
int f1()      // return by value
{
    ...
   return i;
}

int& f2()   // return by reference
{
    ...
   return i;
}
```

```
void main()
{
    ...
   int j;
   j = f1();

    ...
   int k;
   k = f2();
}
```



Return by Reference (Cont.)

- Advantage of return by reference
  - A function that returns a value by reference can be used <u>on the left-hand side of an assignment</u> statement

```
int& array_index(int a[], int i)
{
    return a[i];
}
```

```
int x[] = {1, 2, 3, 4, 5};
int y;
y = array_index(x, 2);
array_index(x, 3) = -16;
```

- Notice on using return by reference
  - The storage cell returned ***MUST*** remain in existence even after the function returns

```
int& f()
{
    int i;          // local variable i
    ...
    return i;       // Caution: i goes out of existence after the function returns
}                   //          (i needs to be declared as "static")
```

Inline Functions

- In C++, a function can be expanded ***inline***
  - I.e., each occurrence of a call to the function is replaced with the code that implements it
  - Inline function vs. macro
    - Macro expansion is performed by a preprocessor at a text (source code) level
    - Inline function expansion is performed by a compiler at a object code level with regard to the semantics of the code
  - Pros and cons
    - Overhead of function calls is avoided → faster execution
    - Size of the executable image becomes larger

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

// Declare function swap as
// inline with keyword inline
inline void swap(int&, int&);

int main() {
   int i = 7, j = -3;
   swap(i, j);
       // to be expanded here
   cout << "i = " << i
        << '\n';
   cout << "j = " << j
        << '\n';
   return 0;
}

void swap(int& a, int& b) {
   int t;
   t = a; a = b; b = t;
}
```

Default Arguments

- C++ allows to specify default values for function parameters
  - If any of arguments is missing in the invocation, its default value is used
  - Only constants can be used as default values
  - All of the parameters without default values must come prior to the parameters with default values

```
void f(int val,
       float s = 12.6,
       char t = '\n',
       string msg = "Error") { ... }

f(14, 48.3, '\t', "OK");
   // val=14, s=48.3, t='\t', msg="OK"
f(14, 48.3, '\t');
   // val=14, s=48.3, t='\t', msg="Error"
f(14, 48.3);
   // val=14, s=48.3, t='\n', msg="Error"
f(14);
   // val=14, s=12.6, t='\n', msg="Error"
f(14, '\t'); // Error: wrong sequence
f();         // Error: no default value
             //        for "val"
```

```
// Invalid sequence of default values
void f(int val = 0,
       float s, // Must be the 1st param.
       char t = '\n',
       string msg = "Error");
```

Function Overloading

- Function's signature consists of
  - Function name
  - The number, data type, and order of parameters
    - ❖ Note that <u>return type is ***NOT*** considered</u>

- C++ supports overloaded functions
  - C++ permits identically named functions within the same scope if they have <u>distinct signatures</u>
  - The compiler determines which version of overloaded functions to be invoked by using matching rules against the signatures of the functions
    - E.g.,

```
void f() and void f(int)
void m(double, int) and void m(int, double)
int s(int) and double s(int)
```

Overloaded Function Example

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

void print(int a);
void print(double a);

void main() {
   int x = 8;
   double y = 8;
   print(x);   // print(int) is invoked
   print(y);   // print(double) is invoked
}

void print(int a) {
   cout << a << '\n';
}

void print(double a) {
   cout << showpoint << a << '\n';
}
```

Function Overloading and Default Arguments

- Note that default parameters do ***NOT*** count towards the parameters that make the function unique

- E.g., suppose calling `printValues(10)` in the following example

```
void printValues(int x);
void printValues(int x, int y=20);
```

  - Tha above example is not allowed, because the compiler cannot disambiguate whether the user wanted
    - `printValues(int)` or
    - `printValues(int, 20)` with the default value

Dynamic Memory Allocation

- C++ provides new operators to allocate memory dynamically at runtime
  - **`new`**: allocates a single memory cell
  - **`new[]`**: allocates an array
  - **`delete`**: frees a single memory cell allocated by **`new`**
  - **`delete[]`**: frees an array allocated by **`new[]`**
  - Cf., function **`malloc`** and **`free`**
    - **`malloc`** and **`free`** are C library functions, while **`new`** and **`delete`** are C++ built-in operators (keywords)
    - ❖ They must ***NOT*** be used together

```
int* int_ptr;
int_ptr = new int;
delete int_ptr;

int_ptr = new int[50];
delete[] int_ptr;
```

Exception Handling

- Exceptions
  - Runtime errors caused by some abnormal condition
    - E.g., `out_of_range` exception: when using out-of-bound index for an array

      `bad_alloc` exception: when `new` is unable to allocate requested memory

- Handling exceptions
  - In C, exceptions are typically handled by testing a return value of a function, or range of a variable, and so on
  - In C++, exceptions can be handled with `try` and `catch` blocks
    - Throwing an exception: an exception can be "thrown" by functions, statements, or the keyword `throw` in the `try` block
    - Catching exceptions: exceptions can be "caught" and handled in the `catch` block

Types of Standard Exceptions

- C++ standard exceptions are defined in the following location
  - Header <exception>: **exception**, **bad_exception**
  - Header <new>: **bad_alloc**
  - Header <typeinfo>: **bad_cast**, **bad_typeid**
  - Header <stdexcept>: **logic_error**s, **runtime_error**s
  - Header <ios>: **ios_base::failure**

```
exception
├── bad_alloc           (thrown by new)
├── bad_cast            (thrown by dynamic_cast when fails with a referenced type)
├── bad_exception       (thrown when an exception doesn't match any catch)
├── bad_typeid          (thrown by typeid)
├── logic_error
│   ├── domain_error
│   ├── invalid_argument
│   ├── length_error
│   └── out_of_range
├── runtime_error
│   ├── overflow_error
│   ├── range_error
│   └── underflow_error
└── ios_base::failure   (thrown by ios::clear)
```

Exception Handling Example (1)

- Exception handling with string functions
  - The string method **`erase`** throws an exception of type **`out_of_range`** when an index into the string is out of bound
  - The methods **`insert`**, **`replace`** and **`substr`** of string may also throw an **`out_of_range`** exception

```
string s = “Test string”;
int index, len;

while (true) {
   cout << "Enter index and " +
           "length to erase: ";
   cin >> index >> len;

   try {
      s.erase(index, len);
   } catch(std::out_of_range& e) {
      cout << e.what();
      continue;
   }  // when
      //   index + len > s.length()

   break;
}
```

Exception Handling Example (2)

```
// defining my_exception based on std::exception
#include <iostream>
#include <exception>
using namespace std;

class my_exception : public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened"; // content of my exception
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (my_exception& e)
  {
    cout << e.what() << '\n';
  }
  return 0;
}
```

Exception Handling Example (3)

```
const int MaxSize = 1000;
float arr[MaxSize];
enum out_of_bounds {underflow, overflow};
          // User defined exceptions can be of any valid type
float& access(int i) {
   if (i < 0) throw underflow;
   if (i >= MaxSize) throw overflow;
   return arr[i];
}

void g() {
   try {
      val = access(k);
   } catch (out_of_bounds t) {
      if (t == underflow) {
         cerr << "arr: underflow...aborting\n";
         exit(EXIT_FAILURE);
      }
      if (t == overflow) {
         cerr << "arr: overflow...aborting\n";
         exit(EXIT_FAILURE);
      }
   }
}
```

Default Exception Handler

- In some cases, the system handles an exception by invoking the default handler function named "**unexpected**"
  - No **try** and **catch** blocks
  - Not caught by user-defined **catch** blocks
- Typically, **unexpected** displays an error message and aborts the program
  - The default handler can be replaced with user defined handler function by

```
...

try {
   s.erase(index, len);
} catch (std::out_of_range& e) {
   cout << "erase: " + e.what();

   throw;
    // Rethrow the exception
}

// no more try-catch blocks are
// defined outside
//   -> handled by the default
//      handler "unexpected"
 ...
```

```
std::set_unexpected(new_handler_func)
```

Other Types of Exception Handling

- Assertions
  - Provide other means of dealing with exceptions especially for debugging
  - Introduce a condition to the system, which must be satisfied (true) for the code to be correct
    - If the condition fails (false), the program terminates with a message
    - Not intended to recover and continue the execution

```
#include <iostream>
#include <cassert>

int main()
{
    assert(2+2==4);
    std::cout << "Execution continues past the first assert\n";
    assert(2+2==5);
    std::cout << "Execution continues past the second assert\n";
}
```

```
Execution continues past the first assert
Assertion failed: 2+2==5,
             file assert_test.cpp, line 8
```

Other Types of Exception Handling (Cont.)

- Signals
  - Exceptions <u>external to the C++ program</u>
    - Also called asynchronous exceptions
    - E.g., **`SIGINT`**: an interrupt request sent to the program by pressing **[Ctrl]-c**
  - Handled by the signal handling library and handler functions

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

void signalHandler(int signum)
{  cout << "Interrupt signal (" << signum << ") received.\n"; }

int main()
{
   signal(SIGINT, signalHandler);   // registers handler for SIGINT
   while(1) { sleep(1); }
}
```