Week 1

NWEN 241
Systems Programming

Alvin C. Valera

alvin.valera@ecs.vuw.ac.nz

Content

- Systems programming
- C/C++ fundamentals
- Identifier scope and namespace
- Arrays

Systems Programming

- Systems programming refers to the implementation of **systems programs** or **software**

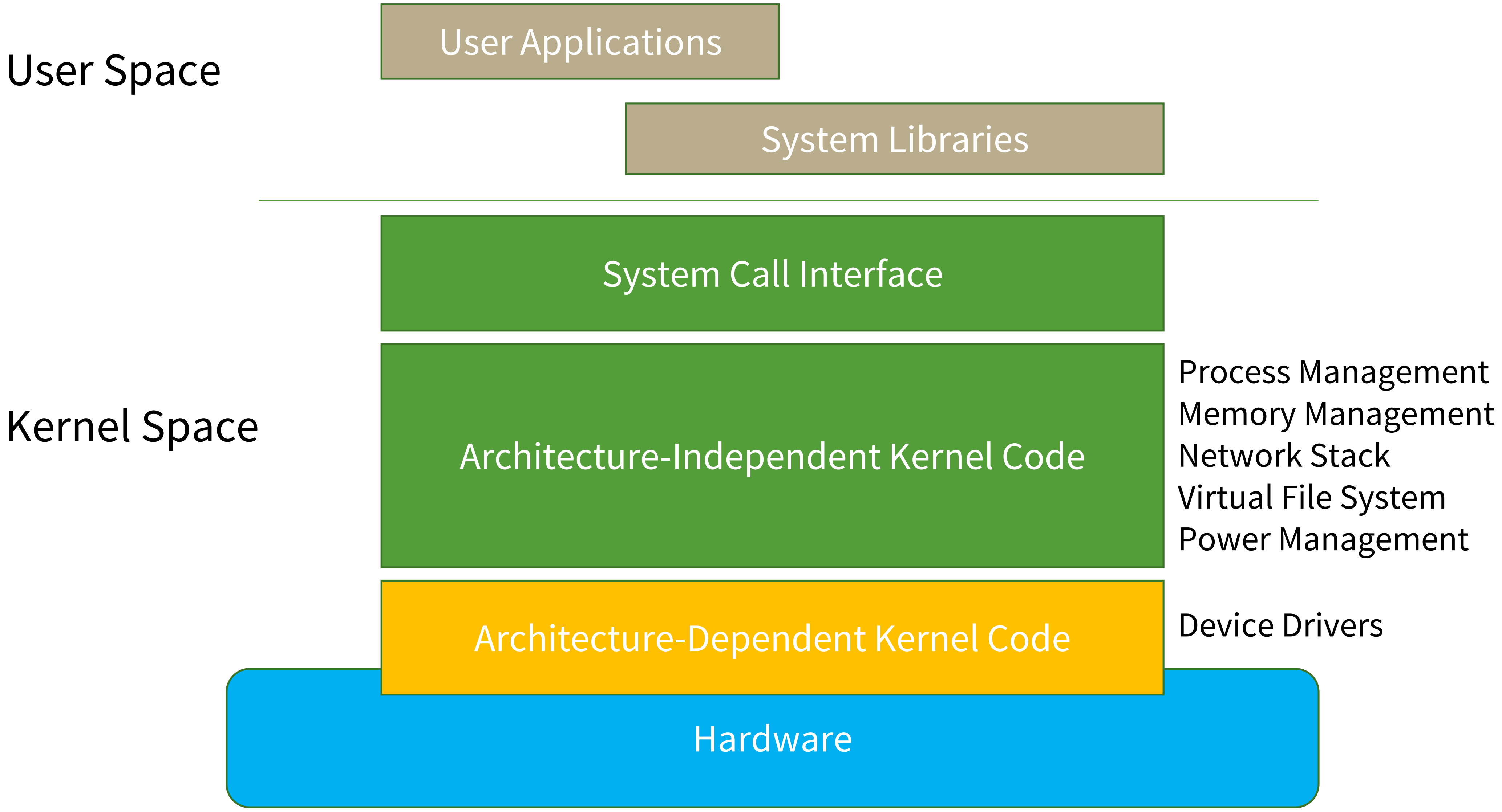
- Systems program / software:
  - Programs that support the **operation** and **use** of the computer system itself
  - Maybe used to support other software and application programs
  - May contain **low-level** or **architecture-dependent** code

- Low-level or architecture-dependent code:
  - Program that directly accesses registers or memory locations
  - Program that uses instructions specific to a computer architecture

Example Systems Programs

- Operating system
- Embedded system software (firmware)
- Device drivers
- Text editors, compilers, assemblers
- Virtual machines
- Server programs
  - Database systems
  - Network protocols

Example: Linux Operating System



Why C/C++?

- C/C++ supports both **high-level** abstractions and **low-level** access to hardware at the same time
- High-level abstractions:
  - User-defined types (structures and classes)
  - Data structures (stacks, queues, lists)
  - Functions
- Low-level access to hardware:
  - Possible access to registers
  - Dynamic memory allocation
  - Inclusion of assembly code

Comparing C, C++ and Java

- C is the basis for C++ and Java
  - C evolved into C++
  - C++ transmuted into Java
  - The “class” is an extension of “struct” in C
- Similarities
  - Java uses a syntax similar to C++ (for, while, . . .)
  - Java supports OOP as C++ does (class, inheritance, . . .)
- Differences
  - Java does not support pointer
  - Java frees memory by garbage collection
  - Java is more portable by using bytecode and virtual machine
  - Java does not support operator overloading
  - . . .

Approach to learning C/C++

- C/C++ fundamentals share many similarities
  - We will be teaching the similar aspects together
  - Assumes knowledge of Java
- Where appropriate, we will show code for pure C and C++ separately
- Key differences are:
  - For standard input and output, C uses `stdio.h` while C++ can use `iostream`
  - C has only one global namespace while C++ allows definitions of namespaces
  - **C programs consists of functions while C++ programs can have functions and classes**

C/C++ Fundamentals

- Identifiers
- Reserved keywords
- Data types
- Operators
- Control flows
- Functions

- C/C++ and Java share many similarities in their fundamentals
- We will talk about the similarities and differences

Identifiers

- Identifier is used to name **macros**, variables, **functions**, **structs**, **unions**, classes, member variables, member functions, and other entities in a computer program

- Java and C/C++ have similar rules for identifiers, except:
  - In C/C++, $ is not allowed

Rules on Identifiers

- An identifier is a sequence of letters and digits
  - The first character must be a letter
    - The underscore character _ counts as a letter
    - Upper and lower case letters are different
- Identifiers may have any length
  - Usually, only the first 31 characters are significant
  - For macro names, only the first 63 characters are significant
- Reserved keywords cannot be used as identifiers!

Examples

`counter`	Valid: consists of letters
`_Temp_variable_2`	Valid: consists of letters and digits
`1myVariable`	Invalid: first character is not a letter
`$steps`	Invalid: $ is not allowed in C/C++
`continue`	Invalid: reserved word

Reserved Keywords

- C reserved keywords

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

Reserved Keywords

- Additional C++ reserved keywords

```
asm            false       public             try
bool           friend      protected          typeid
catch          inline      reinterpret_cast   typename
class          mutable     static_cast        using
const_cast     namespace   template           virtual
delete         new         this               wchar_t
dynamic_cast   operator    throw
explicit       private     true
```

- Newer C++ standards have added even more keywords!

Data Types

- Recall: Java has 8 basic data types which have fixed sizes

Data Type	Size (bytes)
`boolean`	1
`byte`	1
`char`	2
`short`	2
`int`	4
`long`	8
`float`	4
`double`	8

Data Types

- C/C++ data types:

Data Type	Size (bytes)
~~boolean~~	~~1~~
~~byte~~	~~1~~
char	~~2~~ 1
short (short int)	~~2~~ Machine-dependent
int	~~4~~ Machine-dependent
long (long int)	~~8~~ Machine-dependent
long long (long long int)	Machine-dependent
float	~~4~~ Machine-dependent
double	~~8~~ Machine-dependent
long double	10

Integral types

Float types

Data Types

- C++ only data types:

Data Type	Size (bytes)
`bool`	1
`wchar_t`	2 or 4 (Machine-dependent)

Data Type Size

- Sizes of different types
  - Use `sizeof()` to find out
  - As mentioned, some of the types size may vary from machine to machine

- The following rules are always guaranteed:
  - `sizeof(char) = sizeof(bool) = 1`
  - `sizeof(char) < sizeof(wchar_t)`
  - `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`
  - `sizeof(float) <= sizeof(double) <= sizeof(long double)`

Data Types

- C/C++ integral types can either be `signed` or `unsigned`

```
signed int var1;    // Signed integer

unsigned int var2;  // Unsigned integer

int var1;  // If signed or unsigned is not present, default is signed
```

char Data Type

- `unsigned char`: 0 to 255; `signed char`: -128 to 127
- char is meant to hold 1 ASCII character

```
|  0 NUL|  1 SOH|  2 STX|  3 ETX|  4 EOT|  5 ENQ|  6 ACK|  7 BEL|
|  8 BS |  9 HT | 10 NL | 11 VT | 12 NP | 13 CR | 14 SO | 15 SI |
| 16 DLE| 17 DC1| 18 DC2| 19 DC3| 20 DC4| 21 NAK| 22 SYN| 23 ETB|
| 24 CAN| 25 EM | 26 SUB| 27 ESC| 28 FS | 29 GS | 30 RS | 31 US |
| 32 SP | 33  ! | 34  " | 35  # | 36  $ | 37  % | 38  & | 39  ' |
| 40  ( | 41  ) | 42  * | 43  + | 44  , | 45  - | 46  . | 47  / |
| 48  0 | 49  1 | 50  2 | 51  3 | 52  4 | 53  5 | 54  6 | 55  7 |
| 56  8 | 57  9 | 58  : | 59  ; | 60  < | 61  = | 62  > | 63  ? |
| 64  @ | 65  A | 66  B | 67  C | 68  D | 69  E | 70  F | 71  G |
| 72  H | 73  I | 74  J | 75  K | 76  L | 77  M | 78  N | 79  O |
| 80  P | 81  Q | 82  R | 83  S | 84  T | 85  U | 86  V | 87  W |
| 88  X | 89  Y | 90  Z | 91  [ | 92  \ | 93  ] | 94  ^ | 95  _ |
| 96  ` | 97  a | 98  b | 99  c |100  d |101  e |102  f |103  g |
|104  h |105  i |106  j |107  k |108  l |109  m |110  n |111  o |
|112  p |113  q |114  r |115  s |116  t |117  u |118  v |119  w |
|120  x |121  y |122  z |123  { |124  | |125  } |126  ~ |127 DEL|
```

Example

`01000001`

What do you see?

- Interpreted as an integer: 65
- Interpreted as an ASCII character: ‘A’

Variable Declaration

- Similar syntax as Java
- A variable must be declared before it can be used
- A variable may be initialized in its declaration
  - If variable name is followed by an equals sign and an expression, the latter serves as an *initializer*

```
int i = 0, j = 1, k = 2;
char c = 'A';
float f = 1.25;
```

- Possible initializers
  - Constant
  - Expression

Constants

- C/C++ has 2 types of constants:
  - Literal
  - Symbolic
- Literal constant examples:

```
int i = 0;
char c = 'A';
```

  - See https://www.tutorialspoint.com/cplusplus/cpp_constants_literals.htm for more examples
- Symbolic constants can be declared using `const` qualifier or `#define` pre-processor

```
const float PI = 3.14;
```

```
#define PI 3.14
```

Type Casting

- Type casting is a way to convert a variable from one data type to another data type
- C/C++ performs automatic type casting

```
int i = 2;
double d = 2.5;
i = (int)d;        // explicit type casting

i = d;             // d is converted to an int
                   // and then assigned to i
```

static_cast

- C++ provides `static_cast` to perform explicit type conversion

```
static_cast<type>(expression)
```

- Examples:

```
static_cast<int>(7.5)      // evaluates to 7
static_cast<float>(15)     // evaluates to 15.0
static_cast<float>(5/2)    // evaluates to 2.0
```

Another way in C++

- Yet C++ provides another way for explicit type conversion
- Syntax:

```
type(expression)
```

- Examples:

```
int(7.5)      // evaluates to 7
float(15)     // evaluates to 15.0
float(5)/2    // evaluates to 2.5
```

Operators

- Java and C/C++ share many of the built-in operators
  - Arithmetic
  - Assignment
  - Increment/decrement
  - Relational
  - Equality and logical
  - Bitwise
- C/C++ specific operators
  - Pointers and reference related operators (*, &, ->)
  - Others (sizeof, scope, casting)
- In C++, some of the operators can be overloaded

Operator Precedence

- Operator *precedence* determines the sequence in which operators in an expression are evaluated
- *Associativity* determines execution for operators of equal precedence
- Precedence can be overridden by explicit grouping using (and)

Operator Precedence Table (not complete)

Unary operators

Arithmetic operators

Ternary operator

Assignment operators

Operators	Associativity
`()  []  ->  .`	left to right
`!  ~  ++  --  +  -  *` (*type*) `sizeof`	right to left
`*  /  %`	left to right
`+  -`	left to right
`<<  >>`	left to right
`<  <=  >  >=`	left to right
`==  !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	right to left
`,`	left to right

Important Things to Remember

- / denotes integer division
  - 5/2 evaluates to 2 (integer part is used, decimal part is truncated)

- % denotes modulo operation
  - 5%2 evaluates to 1 (the remainder after dividing 5 with 2)

- Increment/decrement operators can only be applied to variables of basic types

```
k++;
counter--;
```

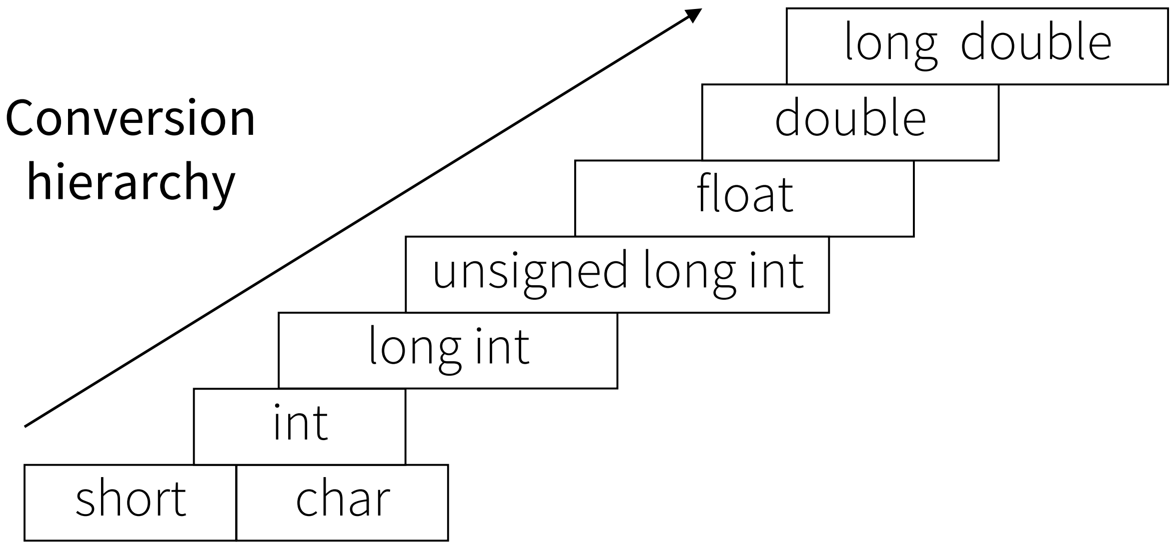
Valid if k and `counter` are variables of basic types

```
777++;
(a + b*c)--;
```

Invalid

“Conversion hierarchy”

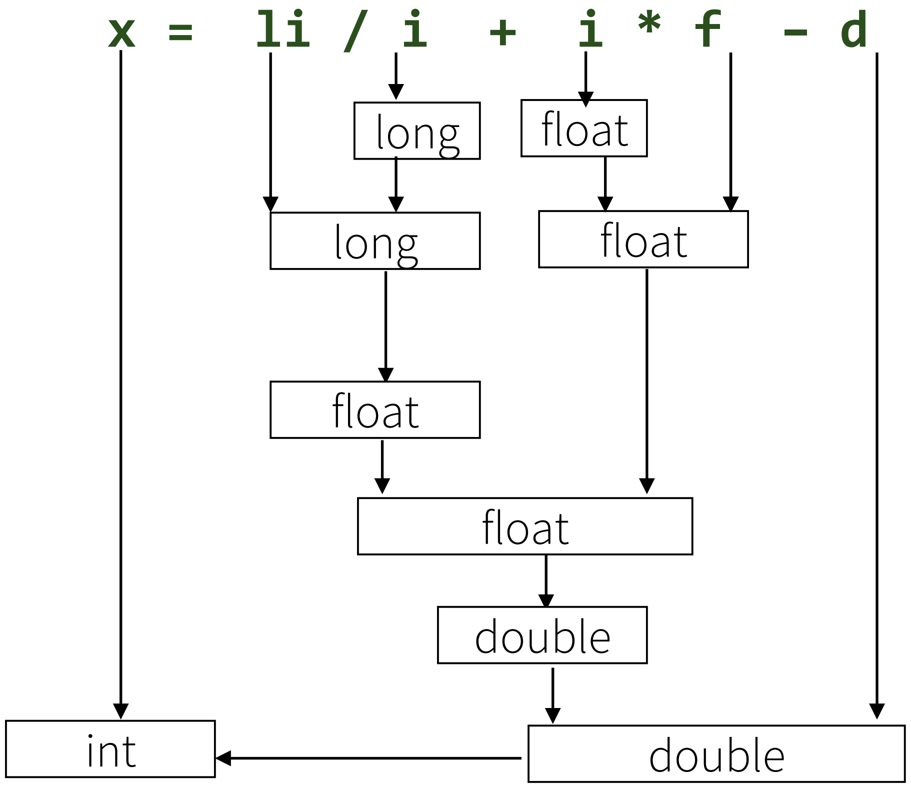
- What happens when operands have different types in an arithmetic expression?
  - **Implicit type conversion is performed:** compiler automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance



Implicit Type Conversion Example

Suppose:

```
int i, x;
float f;
double d;
long int li;
```



The final result of the right hand side expression is converted to the type of the variable on the left of the assignment

Control Constructs

C/C++ Control Constructs

- Control flow
  - If-else
  - Else-if
  - Switch
- Loop
  - While-loop
  - For-loop
  - Do-while-loop
- Same syntax as Java

Differences

Condition in if-else, else-if, while-loop, for-loop and do-while-loop

- In Java, the condition must be an expression that evaluates to boolean
- In C/C++, the condition is an expression that evaluates to any type
  - Considered true if expression evaluates to non-zero value, otherwise false

Break and continue

- In Java, break and continue statements can be labelled or unlabelled
- In C/C++, break and continue statements do not support labels

Example

```
int i = 100;

while (i--) {
    // do stuff
}
```

- Valid in C/C++
- Will generate syntax error in Java
  - Condition inside while-loop should be changed to an expression that will evaluate to boolean type, e.g. `i-- > 0`

Control Constructs

C++ Only Control Constructs

- Exception handling
  - Throw
  - Try-catch
- Almost the same syntax as Java

Functions

- Unlike Java, C/C++ allows functions to exist on their own, i.e., outside any class
  - In C, functions are first-class entities: a C program consists of one or more functions

- A C/C++ program must have exactly one `main` function
- Execution begins with the `main` function

Functions

- General form of a C/C++ **function definition:**

```
return_type function_name ( parameter_list )
{
    body of the function
}
```

Function header

Functions

- Examples

```
void say_hello ( void )
{
    printf("Hello");
}
```

```
int add ( int a, int b )
{
    return a + b;
}
```

Formal parameters

Invoking Functions

- Example function invocations:

```
say_hello();
```

Actual parameters

```
int i = 1, j = 2;
int k = add(i, j);
```

- Before a function can be invoked, either the **function definition** or **function prototype** should have been declared prior to the invocation
- **Function prototype** – declaration specifying the return type, function name, and list of parameter types

```
return_type function_name ( parameter_types_list );
```

Function Prototype

- Examples

```
void say_hello ( void );
```

```
int add ( int a, int b );
```

- No need to provide identifiers to input parameters, the types of the input parameters are sufficient

```
int add ( int, int );
```

Program Structure

- A typical C/C++ program consists of
  - 1 or more **header** files
  - 1 or more C/C++ **source** files

```
#include <stdio.h>

int main(void)
{
   printf("Hello world\n");

   return 0;
}
```

Preprocessor directive to include `stdio.h` header file which contains `printf` function *prototype*

`main` function *definition*, invoking `printf` to display “Hello, world”, and return 0

Hello world using pure C

Program Structure

- A typical C/C++ program consists of
  - 1 or more **header** files
  - 1 or more C/C++ **source** files

```
#include <iostream>

int main(void)
{
    std::cout << "Hello world\n";

    return 0;
}
```

Preprocessor directive to include `iostream` header file which contains std::`cout` function *prototype*

`main` function *definition*, invoking `std::cout` to display “Hello, world”, and return 0

Hello world using C++

Header File Inclusion

```
#include <filename>
```

- Include file named `filename`
- Preprocessor searches for file in pre-defined locations

```
#include "filename"
```

- Include file named `filename`
- Preprocessor searches for file in current directory first, then in locations specified by programmer

Header Files

- A header file usually contains function prototypes, constant definitions, type definitions, etc.

- Which header file to include?
  - Include header files that contain the function prototype, constant definition, type definition, etc., used in your program
  - Tutorial 1 will introduce the header files in the standard C and C++ libraries

Another example program (pure C)

```c
/* Program to calculate the area of a circle */
#include <stdio.h>
#define PI 3.14

float sq(float);

int main(void)
{
   float radius, area;

   /* Ask user to input */
   printf("Radius = ");
   scanf("%f", &radius);

   area = PI * sq(radius);
   printf("Area = %f\n", area);
   return 0;
}

float sq(float r)
{
   return (r * r);
}
```

Preprocessor directives

Function prototype

main
function

Function definition

Another example program (C++)

```
/* Program to calculate the area of a circle */
#include <iostream>
#define PI 3.14

float sq(float);

int main(void)
{
   float radius, area;

   /* Ask user to input */
   std::out << "Radius = ";
   std::in >> radius;

   area = PI * sq(radius);
   std::out << "Area = " << area << "\n";
   return 0;
}

float sq(float r)
{
   return (r * r);
}
```

Preprocessor directives

Function prototype

main function

Function definition

Macro Substitution

```
#define name replacement
```

- Subsequence occurrences of `name` will be replaced by `replacement`

Function-like Macro

- Can abuse macro substitution to define **function-like** macros
- To define a function-like macro, just append `()` to the macro name
- Example:

```
#define READ_CHAR()     getchar()
```

- Can be invoked like a regular function:

```
…
int c = READ_CHAR();
…
```

Function-like Macro

- Just like functions, function-like macros can take arguments
  - Insert comma-separated parameter names between (and)
  - Parameter names must be valid identifiers

```
#define max(X, Y) ((X) > (Y) ? (X) : (Y))
```

- Invoke just like normal functions

```
z = max(1, 3);
```

```
z = ((1)>(3)?(1):(3));
```

This expression evaluates to **3**

Problems with Function-like Macros

- Suppose:

```
#define SQ(X)      X * X
```

- Then:

```
(int)SQ(r);        →   (int)r * r;
SQ(r1 + r2);       →   r1 + r2 * r1 + r2;
```

- Solution: enclose individual variables with (), including the whole replacement text

```
#define SQ(X)      ((X) * (X))
```

Problems with Function-like Macros

- Suppose:

```
#define SQ(X)       ((X) * (X))
```

- Then:

```
(int)SQ(r);          →   (int)((r) * (r));
SQ(r1 + r2);         →   ((r1 + r2) * (r1 + r2));
```

Problems with Function-like Macros

- Suppose:

```
#define SQ(X)       ((X) * (X))
```

- How about these:

```
SQ(++r);
```

→

```
((++r) * (++r));
```

`r` incremented twice

```
SQ(f());
```

→

```
((f()) * (f()));
```

`f()` invoked twice

Be careful when defining and calling function-like macros!

Identifier Scope

- Identifier scope refers to parts of the program where an identifier is accessible or visible

- **Local**: identifiers declared within a function or block - only visible inside the block
- **Global**: identifiers declared outside functions - visible from the line of declaration to the end of file

Example: local scope

```
int func(float a, int b)
{
  int i;
  double g;

  for (i = 0; i < b; i++) {
    double h = i*g;
    // loop body - may access a, b, i, g, h
  } // end of for-loop
  // func body - may access a, b, i, g
} // end of func()
```

`i` is visible from this point to end of `func`

`g` is visible from this point to end of `func`

h is only visible from this point to end of loop!

Example: global scope

```
#include <stdio.h>

float x = 1.5; /* Definition - extern class - global */

void show (void)
{
    printf("%f\n", x); /* Access global x */
}

int main (void)
{
    printf("%f\n", x); /* Access global x */
    show();
    return 0;
}
```

What if x is defined after main and you want to use it?

Example: global scope

```c
#include <stdio.h>

void show (void)
{   extern float x;
    printf("%f\n", x); /* Access global x */
}

int main (void)
{   extern float x;
    printf("%f\n", x); /* Access global x */
    show();
    return 0;
}

float x = 1.5; /* Definition - extern class - global */
```

Difficulties with Global Identifiers

- When a header file is included in a program, all global identifiers in the header file also become global identifiers in the program
- When program has global identifiers with same name as in header file, compiler will generate an error (e.g., “identifier redefined”)

- Can be solved using the **namespace** mechanism

Namespace

- Only available in C++
- General syntax:

```
namespace namespace_name
{
    members
}
```

- Members can be constants, variables, functions, classes, or another namespace

Example:

```
namespace myns
{
   const int N = 100;
   int count = 0;
   void printResult();
}
```

Namespace

- The scope of a namespace member is **local** to that namespace
- Member identifier is not visible outside its namespace

Two ways to access a namespace member outside its namespace:

- Use `namespace_name::identifier` syntax
- Use the `using` keyword to access specific or all members of a namespace

Example

```
namespace myns
{
    const int N = 100;
    int count = 0;
    void printResult();
}
```

Scope resolution operator

Expression to access N:

```
myns::N
```

Expression to invoke `printResult()`:

```
myns::printResult();
```

Example

```
namespace myns
{
    const int N = 100;
    int count = 0;
    void printResult();
}
```

Make all members visible

```
using namespace myns;
```

Expression to access N:

```
N
```

Expression to invoke `printResult()`:

```
printResult();
```

Example

```
namespace myns
{
   const int N = 100;
   int count = 0;
   void printResult();
}
```

Make a specific member visible

```
using myns::N;
```

Expression to access N:

```
N
```

Example

```cpp
#include <iostream>

int main(void)
{
   std::cout << "Hello world\n";

   return 0;
}
```

```cpp
#include <iostream>

using namespace std;

int main(void)
{
   cout << "Hello world\n";

   return 0;
}
```

Arrays

- An array is a collection of data that holds a **fixed** number of data (values) of the **same type**
- In C/C++, arrays and pointers are closely related concepts
  - An array name by itself is treated as a *constant pointer*
- We distinguish between two types of arrays:
  - One-dimensional arrays
  - Multi-dimensional arrays
    - The C/C++ language places no limits on the number of dimensions in an array, though specific implementations may

Declaring Arrays

- Declaring arrays in C/C++ differs slightly compared to Java
- Syntax for **declaring** a one-dimensional array:

```
data_type array_name[size];
```

- Example:
  - We declare an array named `data` of `float` type and size `4` as:

```
float data[4];
```

  - It can hold 4 floating-point values

- The **size** and **type** of arrays <u>**cannot**</u> be changed after their declaration!

Initializing Arrays

- Arrays can be initialized **one-by-one**
- For example:

```
float data[4];
data[0] = 22.5;
data[1] = 23.1;
data[2] = 23.7;
data[3] = 24.8;
```

- In the case of large arrays this method is <u>inefficient</u>

Initializing Arrays

- Arrays can be also initialized when they are **declared** (just as any other variables):

```
float data[4] = {22.5, 23.1, 23.7, 24.8};
```

- An array may be **partially initialized**, by providing fewer data items than the size of the array

```
float data[4] = {22.5, 23.1};
```

  – The remaining array elements will be automatically initialized to zero

- If an array is to be completely initialized, the dimension (size) of the array is not required

```
float data[] = {22.5, 23.1, 23.7, 24.8};
```

  – The compiler will automatically size the array to fit the initialized data

Determining Size of Array

- The size of an array can be determined using the `sizeof()` operator

- It will return the *number of* ***bytes*** *the array "occupies" in the memory*

- To determine the number of elements in the array, the <u>**returned**</u> value must be <u>**divided**</u> by the <u>**number of bytes**</u> reserved for the <u>**data type**</u> **!**

Determining Size of Array (pure C)

```
int data[] = {1, 2, 3, 4, 5};
int bytes, len;

/* Print number of bytes used by array */
bytes = sizeof(data);
printf("Bytes used: %d\n", bytes);

/* Print number of elements or items in array */
len = sizeof(data)/sizeof(int);
printf("Number of items: %d\n", len);

/* To traverse array, use number of elements as limit */
for (int idx = 0; idx < len; idx++) {
      /* do some stuff on element data[idx] */
}
```

Determining Size of Array (C++)

```
int data[] = {1, 2, 3, 4, 5};
int bytes, len;

// Print number of bytes used by array
bytes = sizeof(data);
std::cout << "Bytes used: " << bytes << "\n";

// Print number of elements or items in array
len = sizeof(data)/sizeof(int);
std::cout << "Number of items “ << len << "\n";

// To traverse array, use number of elements as limit
for (int idx = 0; idx < len; idx++) {
      // do some stuff on element data[idx]
}
```

Arrays and C Strings

- A character array that contains ASCII characters terminated by the null character '\0' is a C string variable
- Such an array can be initialized using methods 1 and 2

```
char str[10];
str[0] = 'H';
str[1] = 'e';
str[2] = 'l';
str[3] = 'l';
str[4] = 'o';
str[5] = ' ';
str[6] = '!';
str[7] = '\0';
```

```
char str[10] = {
  'H', 'e', 'l',
  'l', 'o', ' ',
  '!', '\0' };
```

Arrays and C Strings

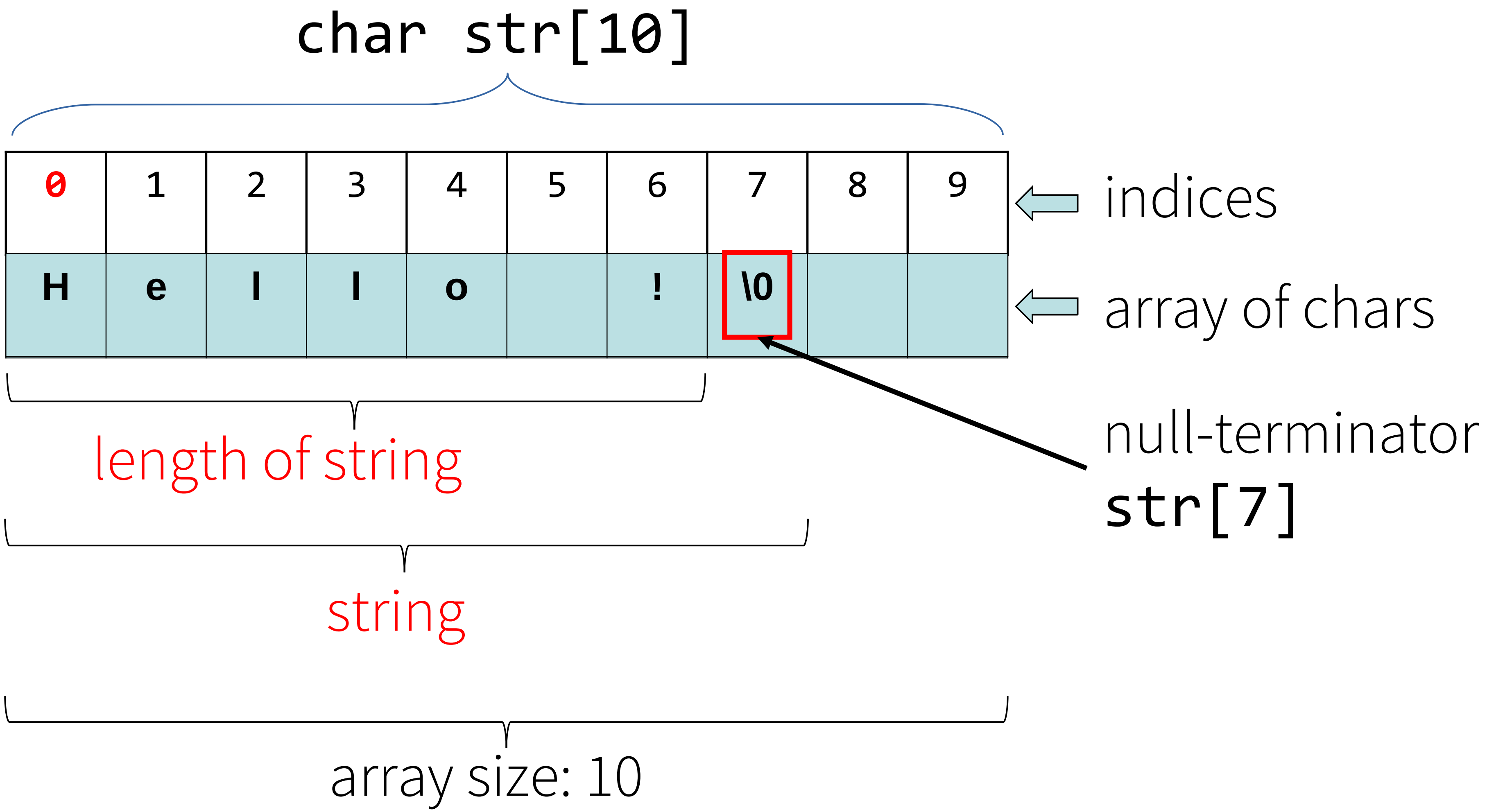
- Another way to initialize a character array to hold a string variable

```
char str[10] = "Hello !";
```

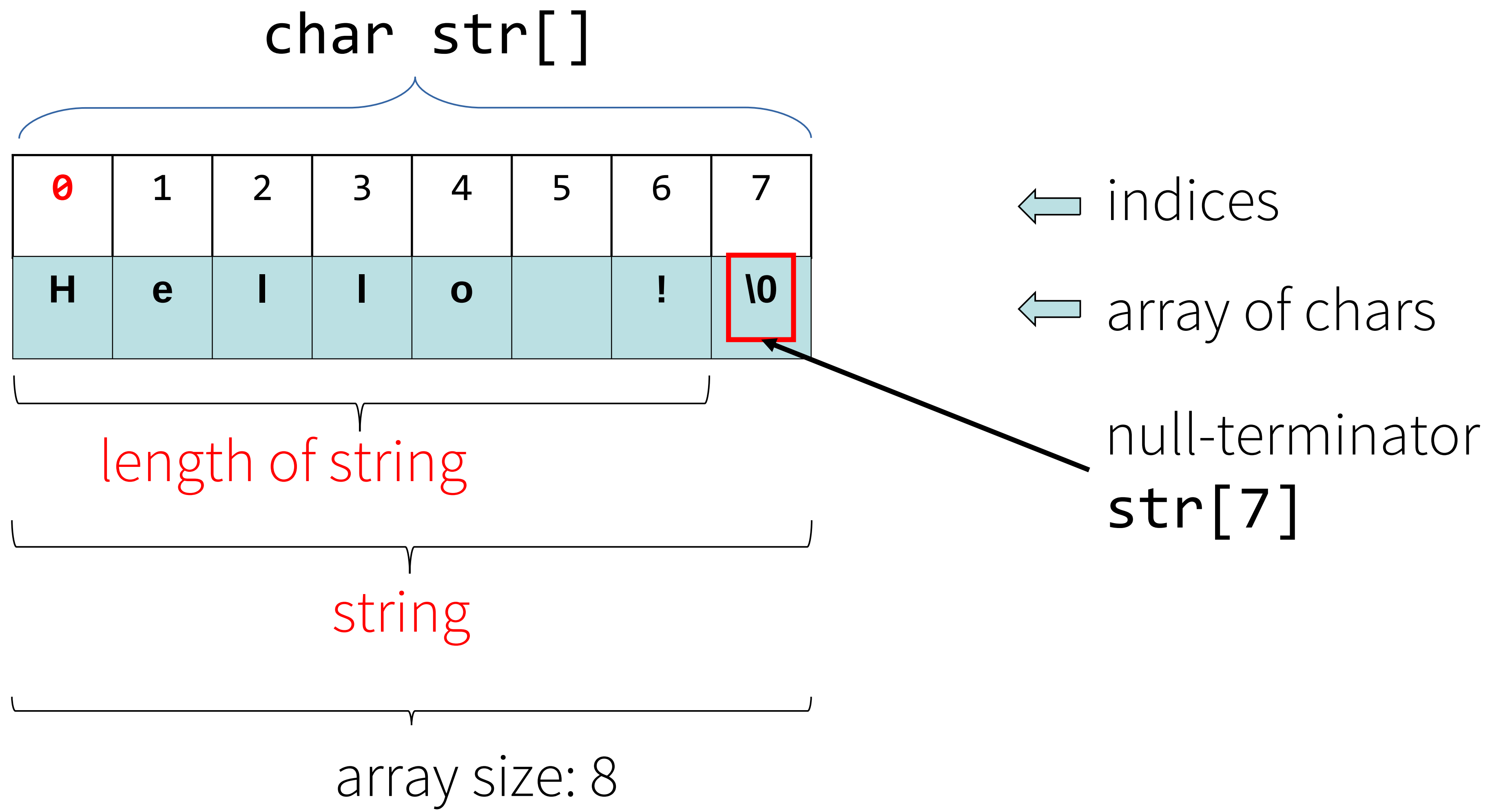
```
char str[] = "Hello !";
```

What's the difference between the two?

```c
char str[10] = "Hello !";
```



```
char str[] = "Hello !";
```



2D Arrays

- Declaring a char array with 3 rows and 5 columns

```
char two_d[3][5];
```

  – The array can hold 15 char elements
- Accessing a value

```
char ch;
ch = two_d[2][4];
```

- Modifying a value

```
two_d[0][0] = 'x';
```

- The array can be initialized in one of the following ways

```
int two_d[2][3] = {{5, 2, 1}, {6, 7, 8}};
int two_d[2][3] = {5, 2, 1 , 6, 7, 8};
int two_d[][3] = {{5, 2, 1}, {6, 7, 8}};
```

  – The number of columns must be explicitly stated. The compiler will find the appropriate amount of rows based on the initializer list

3D Arrays

- Declaring a three-dimensional (3D) array

```
float three_d[2][4][3];
```

  - Here, three_d can hold 24 elements. Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements.

- Initializing a 3D array

```
int test[2][3][4] = {
        {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
        {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}
};
```

Next Lecture

- User-defined types

- C++ classes