Object Oriented Programming with C++

2024 Spring Semester

21 CST H3Art

Chapter 3 Basic Concepts

- **Tokens（标记）**: the smallest individual units in a program:
  - Keywords （关键字）
  - Identifiers （标识符）
  - Constants （常量）
  - Reference（引用）
  - Strings （字符串）
  - Operators （运算符）
- **Keywords（关键字）**: A reserved word that has special meaning in C++ and can not be used as a programmer-defined identifier.

	C++	**keywords**	
asm	**double**	new	**switch**
auto	**else**	operator	template
break	**enum**	private	this
case	**extern**	protected	throw
catch	**float**	public	try
char	**for**	**register**	**typedef**
class	friend	**return**	**union**
const	**goto**	**short**	**unsigned**
continue	**if**	**signed**	virtual
default	inline	**sizeof**	**void**
delete	**int**	**static**	**volatile**
do	**long**	**struct**	**while**
Added by ANSI C++			
bool	export	reinterpret_cast	typename
const_cast	false	static_cast	using
dynamic_cast	mutable	true	wchar_t

	C++	keywords	
explict	namespace	typeid	

*Note: The ANSI C keywords are shown in **bold** face.*

- **Identifiers（标识符）**: naming program element
  - Identifiers are made up of:
    - letters (A-Z, a-z)
    - digits(0-9)
    - the underscore character(_)
    - and **must begin with a letter or underscore**
  - Example:
    - Sum_of_squares $\rightarrow \surd$
    - J9 $\rightarrow \surd$
    - 40Hours $\rightarrow \times$
    - _22A $\rightarrow \surd$
    - G et Data $\rightarrow \times$
    - Box-1 $\rightarrow \times$
    - Cost_in_$ $\rightarrow \surd$
    - namespace $\rightarrow \times$
- **Constants（常量）**: fix values that do not change during the execution of a program
  - **Literal constant（字面常量）**： Any constant value written in a program, like `10`, `3.14`, `'a'`.
  - **Symbolic constant（命名常量）**: A location in memory, referenced by an identifier, that contains a data value that cannot be changed:
    - Using the qualifier **const**
    - Defining a set of integer constants using **enum** keyword
  - **Qualifier const（const 说明符）** allow us to create typed constants
    - Called as "**declared constant**" or "**named constant**"
    - A const **must be initialized** when declared
    - A const is **local to the file where it is declared**
      - Use **extern** to make it can be referenced from another file
  - **Const with pointers**
    - **Pointer to a constant（指向常量的指针）**:

      ```
      const int * pi;
      ```

      - Pointer `pi` points to a constant
      - Cannot change **the contents of what it points to** through the pointer
      - The **value of the pointer** itself **can be changed**
      - Example:

```
int a = 1, b = 2;
const int c = 3;
const int *pi;
pi = &a; // OK
*pi = 10; // Err
a = 10; // OK
pi = &c; // OK
*pi = 100; // Err
c = 100; // Err
```

- **Constant pointer（常指针）**：

```
int a;
int * const pi = &a;
```

  - The pointer itself is a constant，**must be initilaized**
  - Contents of **what it points to can be changed**
  - Example:

```
int a = 1, b = 2;
const int c = 3;
int * const pi = &a;
*pi = 10; // OK
pi = &b; // Err
int * const pint_c = &c; // Err
*pi = 20; // OK
```

- **Constant pointer points to a constant（指向常量的常指针）**：

```
const int * const pi = &a;
```

  - The value of the pointer can not be changed
  - Cannot change the contents of what it points to through the pointer
  - Example:

```
int a = 1, b = 2;
const int c = 3;
const int *const pi = &a;
*pi = 10; // Err
pi = &b; // Err
```

- **Enumerated data type （枚举）**
  - Enumerated data type example:

```
enum shape{circle, square, triangle};
shape myshape;
myshape = circle;
```

    - By default, the **enumerators（枚举常量）** are assigned integer values **starting with 0**
    - An enumerated value can be used in place of an **int value**:

```
int c = circle; // equals to int c = 0;
```

        - We can over-ride the default by explicitly assigning integer values:

```
enum shape{circle, square = 4, triangle = 8};
enum shape{circle = 6, square, triangle};
```

- **The type void（空类型）**
    - Specify the return type of a function

```
void func();
```

    - Indicate an empty argument list to a function

```
void func(void);
```

    - Generic pointer（通用指针）/ `void*`

```
void * gp;
int i;
int * ip = &i;
gp = ip; // OK
ip = gp; // Err
ip = (int *)gp; // OK
```

- **Reference Variables（引用）**
    - A reference variable provides an **alias(alternative name)** for a previously defined variable.
    - A reference variable is created as following:

```
data_type reference_name = variable_name
```

        - A reference must be **initialized at the time of declaration**
        - Reference to pointer is legal:

```
int * a;
int * &P = a; // Pointer's reference
int b;
P = &b; // Let P become the reference of b's pointer(address)
int &m = *a; // Let m become the reference of a
```

        - Reference to array is **illegal**:

```
int n[10];
int & x[10] = n; // Err
int & x=n[9];
```

    - Pass arguments by **references**:

```
void f(int &x){
  x = x + 10
}

int main(){
  int m = 10;
  f(m);
  cout << m << endl;
  return 0;
}
```

- Return by **reference**
  - For a function returns a reference, what returned can be:
    - **independent reference**
    - **array element**
    - **static variable**
    - **variable pointed by a pointer**
  - A returned reference must be valid in the calling program

```
int & get_reference(int x){
  return x; // Err
}
```

- **Operators in C++**
  - All C operators are valid in C++
  - Additional operators include:
    - `::`
    - `::*`
    - `::->`
    - `.->`
    - `delete`
    - `new`
    - `endl`
    - `setw`
  - Scope resolution operator(`::`)
    - The global version of a variable can be accessed from within the inner block by using the scope resolution operator `::` as:

```
:: variable_name
```

  - `new` & `delete` operators
    - Malloc & free
      - Function prototype:

```
void * malloc(size_t size);
void free(void *p);
```

      - Shortcomes:
        - User needs to **calculate the size** of the data object

        - The data type of the return value is `void *`.User **need to use a type cast**
        - After free the memory space, the function does **not set the pointer to** `NULL`
    - Operator `new` dynamically allocates memory to hold a data object of type data_type and returns the address of the object:

```
pointer_variable = new data_type;
pointer_variable = new data_type(value);
pointer_variable = new data_type[size];
```

        - when creating multi-dimensional arrays with `new`, **all the array sizes must be supplied**. The **first dimension may be a variable** whose value is supplied at runtime:

```
int *p = new int[3][4][5];
int *p = new int[m][4][5]; // Both available
```

        - If sufficient memory is not available for allocation, `new` returns a **null pointer** (the value of a **null pointer is 0**)
        - It automatically **computes the size** of the data object.
        - It automatically returns the **correct pointer type**.
        - It is possible to **initialize the object** while creating the memory space.
    - Operator `delete` releases the memory space allocated by `new` for reuse:

```
delete pointer_variable;
delete []pointer_variable;
```

        - The operand must be a pointer returned by `new`
        - `delete` assign **NULL** to the pointer after release the memory pointed by the pointer
    - `new` and `delete` can be **overloaded**
- **Manipulators（控制符）**
    - Many manipulators, user can **define his own manipulators**
        - `endl`: cause a linefeed to be inserted
        - `setw`: specify a common field width for display
    - Example:

```
#include <iomanip>
#include <iostream>

using namespace std;

int main(){
  int Basic = 950, Allowance = 95, Total = 1045;
  cout << setw(10) << "Basic" << setw(10) << Basic << endl
       << setw(10) << "Allowance" << setw(10) << Allowance << endl;
  return 0;
}
```

    Output is **right aligned**:

```
    Basic       950
Allowance        95
```

- **Type conversions（类型转换）**
  - Use `typedef` to create an identifier of a type

```
typedef type_name alias

// Example
typedef int INT;
typedef double (*DblArrPtr)[10];

INT anInt; // Type: INT (aka int)
DblArrPtr dblArr; // Type: DblArrPtr (aka double (*)[10])
```

  - **Implicit conversions（隐式转换）**
    - Expressions with operands of different data type：**"smaller" type converts to the "wider" type**
    - Float point value to an int variable: **discard the fractional part**
    - Integer value to a float variable, **save in exponential form without changing the value**
    - `int`、`short`、`long` value to `char` variable, only send **the value saved in the lower 8 bit**
    - `char`、`short` → `int` → `unsigned` → `long` → `double` ← `float`
    - It will happen when **actual arguments（实参）** pass value to **formal arguments（形参）**
  - **Explicit type conversion（显式转换）**
    - Explicit type conversion using type cast operator
      - C notation: `(type_name) expression` or `(type_name) (expression)`
      - C++ notation: `type-name (expression)`
    - The functional-call notation can not be used if the type is compound:

```
void *q;
int *p;

p = int *(q); // Err
p = (int *)q; // OK
```

    - New **cast operators** added in C++
      - **const_cast**: `const_cast<type_name>(expression)`
        - removing the const_ness of an object
      - **static_cast**:
        - Conversion between build-in data types
        - Convertion from derived class object pointer or reference to base class object pointer or reference
        - Cannot do conversion between **independent types**
          - Example:

```
float fnum = 5.8;
int inum = static_cast<int>(fnum); // OK

float *pf = &fnum;
int *n = static_cast<int*>(&fnum); // Err, conversion betweeen independent types

void *p = &fnum;
float *dp = static_cast<float*>(p); // OK, find back the value stored in void* po
```

- **dynamic_cast**: 安全的父类指针或引用和子类指针或引用之间的转换（will be mentioned in OOP lecture）
- **reinterpret_cast**: 将一个指针类型转换成其他指针类型，将一个整数类型转换成指针类型或将指针类型转换成整数类型。常用于在函数指针类型之间进行转换。
  - Example:

```
int num;
int *pnum = &num;

char *pc = reinterpret_cast<char *>(pnum);
```

- **Expressions（表达式）**
  - **Arithmetic expressions（算术表达式）**

Annotation	Description
++	Unary plus
--	Unary minus
+	Addition
-	Subtraction
*	Multiplication
/	Floating-point division (floating-point result) Integer division (No fraction part, gives only the integer quotient（商）
%	Modulus (Used only with integers)

  - **Assignment expressions（赋值表达式）**
    - Assignment expression has a value, the value is saved in a temporary variable and can be assigned to another variable.
    - Example:

```
int y = 10;
int x = (y = 10); // OK
x = y = 100;
float a = b = 12.3; // Err, b must be defined previously
x = (y = 50) + 10;
```

  - **Bool Expressions（布尔表达式）**

        - Relational expressions: <, >, <=, >=, ==, !=
        - Logical expressions: &&, ||, !
    - **Bitwise expressions（位运算表达式）**
    - **Precedence（优先级）**
        - arithmetic > relational > logical > assignment
        - The **unary operations** assume higher precedence
- **Control structures（控制结构）**
    - if-else
    - switch-case
    - while
    - do-while
    - for
    - break and continue
    - go to
    - return
    - try-catch