> 由 KiOii (*_EM_Cpper*)整理。 (KiOii (*_EM_Cpper*) makes this note.)

Part I: The Basics

Chapter 2. Variables and Basic Types

Section 2.1 Primitive Built-in Types

Section 2.2 Variables

Section 2.3 Compound Types

Section 2.4 const Qualifier

Section 2.5 Defining Our Own Data Structures

Chapter Summary

Defined Terms

2.1 Primitive Built-in Types

- *include :* ***arithmetic types*** and ***void***
  - *arithmetic-include:* ***characters***、***integers***、***boolean***、***float-point numbers***
  - *void:* **no value,most as the return type**

2.1.1 Arithmetic Types

- *include:* ***integral types***(include ***character*** and ***boolean types***) and ***floating-point types***

Type	Meaning	Minimum Size
bool	boolean	NA
char	character	8 bits
wchar_t	wide character	16 bits
char16_t	Unicode character	16 bits
char32_t	Unicode character	32 bits
short	short integer	16 bits
int	integer	16 bits
long	long integer	32 bits
long long	long integer	64 bits
float	single-precision floating-point	6 significant digits
double	double-precision floating-point	10 significant digits
long double	extended-precision floating-point	10 significant digits

long long: introuced by C++11 standard.

byte

- *defind:* **The smallest chunk of addressable memory**

Signed and Unsigned Types

- *include:* **expect for bool** and **the extended charactor type,**the **integeral types** may be **signed** or **unsigned.**
- *three-charactor types:* ***char***、***signed char***、***unsigned char***
  - *wraning:* ***char*** is not the same type as ***signed char***

    - *wraning:* due to compiler,***char*** is one of the ***signed char*** and ***unsigned char***[1]
    - *wraning:* **The standard does not define how signed types are represented**

- *advice:* Use ***double*** for floating-point computations

2.1.2 Type Conversions

what happens depends on the range of the values that the types permit:

- When we assign one of the nonbool arithmetic types to a bool object, theresult is false if the value is 0 and true otherwise.
- When we assign a bool to one of the other arithmetic types, the resulting value is 1 if the bool is true and 0 if the bool is false.
- When we assign a floating-point value to an object of integral type, the value is truncated. The value that is stored is the part before the decimal point.
- When we assign an integral value to an object of floating-point type, the fractional part is zero. Precision may be lost if the integer has more bits than the floating-point object can accommodate.
- If we assign an out-of-range value to an object of unsigned type, the result is the remainder of the value modulo the number of values the target type can hold. For example, an 8-bit unsigned char can hold values from 0 through 255, inclusive. If we assign a value outside this range, the compiler assigns the remainder of that value modulo 256. Therefore, assigning –1 to an 8-bit unsigned char gives that object the value 255.
- If we assign an out-of-range value to an object of signed type, the result is undefined. The program might appear to work, it might crash, or it might produce garbage values.

Expressions Involving Unsigned Types

both unsigned and int: ***int*** is converted to ***unsigned***

wraning: **unsigned never be less than 0**

2.1.3 Literals

- *wraning:* **Every literal has a type**

Integer and Floating-Point Literals

Integer Literals

- *notation:* **decimal、octal、hexadecimal**
    - *decimal:* such that **20** **signed**
    - *octal:* begin with 0,such that **024** **unsigned or signed**
    - *hexadecimal:* begin with 0x,such that **0x14** **unsigned or signed**

Floating-Point Literals

- notation: 3.14159、3.14159E0、0.、0e0、.001

Character and Character String Literals

```
'a'           // character literal
"Hello World!" // string literal
```

string literal: **array of constant chars**

- *wraning:* **The compiler appends a null character ('\0') to every string literal.**

```
// multiline string literal
std::cout << "a really,really long string literal "
          << "that spans two lines" << std::endl;
```

Escape Sequences

No Visible Imge Character

- *such that:* **backspace** or **control characters**
- *sort:* **nonprintable**、**escape sequence**
- *escape sequence*
  - o

newline	\n	horizontal tab	\t	alert (bell)	\a
vertical tab	\v	backspace	\b	double quote	\"
backslash	\\	question mark	\?	single quote	\'
carriage return	\r	formfeed	\f		

\7 (bell)	\12 (newline)	\40 (blank)
\0 (null)	\115 ('M')	\x4d ('M')

- wraning: **Note that if a \ is followed by more than three octal digits, only the first three are associated with the \.**
- wraning:**\x uses up all the hex digits following it**

Specifying the type of a Literal

```
L'a'      // wide character literal, type is wchar_t
u8"hi!"   // utf-8 string literal (utf-8 encodes a Unicode character in 8 bits)
42ULL     // unsigned integer literal, type is unsigned long long
1E-3F     // single-precision floating-point literal, type is float
3.14159L  // extended-precision floating-point literal, type is long double
```

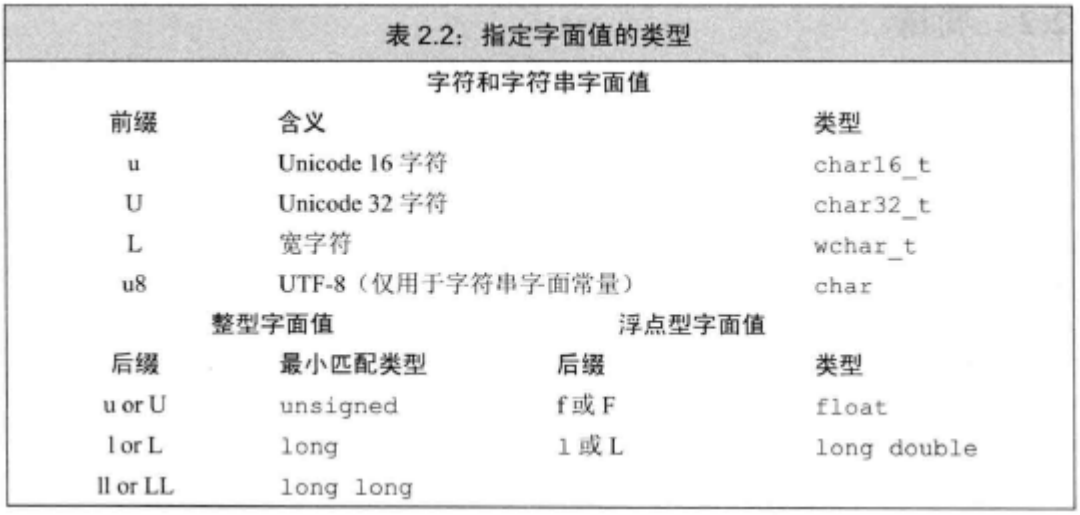

表 2.2：指定字面值的类型

字符和字符串字面值

前缀	含义	类型
u	Unicode 16 字符	char16_t
U	Unicode 32 字符	char32_t
L	宽字符	wchar_t
u8	UTF-8（仅用于字符串字面常量）	char

整型字面值		浮点型字面值	
后缀	最小匹配类型	后缀	类型
u or U	unsigned	f 或 F	float
l or L	long	l 或 L	long double
ll or LL	long long		

Boolean and Pointer Literals

Boolean Literals

- *two:* **true** and **false**

Pointer Literals

- **nullptr**

2.2 Variables

> - Variables in C++ has a **type**
> - The type dermines the **size and layout** of the variables's memory

2.2.1 Variable Definitions

- *doing:* **type specifier + a list of one or more variable**
- *such that:* `int sum = 0,value;`

Initializers

- *define:* An **object** that is initialized **gets the specitfied value** the moment it is **created.**
- *such that:* `int sum = 0,value = sum;`
- *wraning:* **initialization** and **assignment** are **different** operation in C++

List Initialization

```
int var = 0;
int var = {0};
int var{0};          // C++11
int var(0);
```

- *how initialize:* use **{}**

```
int var {1.1}; // compile error
```

Default Initialization

- **default value**(*depend on the **type** and depend on **where** the variable is defined*)
- *specified:* the object of **built-in type** depend one **where** it is deined
  - *outside any function body:* **0**
  - *inside function body:* **uninitialized**

2.2.2 Variable Declarations and Definitions

> *separate compilation:* split our programs into **several files**, **each** of which can be **compiled independently**.

> To support separate compilation,C++ distinguishes between **declarations and definitions**.

declaration

- *do-what:* **makes a name known to the program**
- *use:* use a name defined elsewhere **includes a declaration** for that name
- *code-look:* `extern int i; //declares but does not define i`

definition

- *do-what:* **create the associated entity**
- *code-look:* `int i;// declares and defines i`
- *code-look:* `extern int i = 10;// initialization ignore extern`

Static Type

> C++ is a **statically typed language**,which means that types are checked at compile time.
>
> The process by which types are checked is referred to as **type checking.**[1]

2.2.3 Identifiers

- *include:* **letters(a,b,c,...)、digits(0,1,2,...)、underscore character(_)**
- *wraning:* `__、_A // 用户不允许使用连续两个下划线 和 不允许下划线紧靠大写字母开头`

Conventions for Variable Names

- meaning
- variable: index,not Index or INDEX
- class: Index、Sales_item
- student_loan,studentLoan,not studentloan

2.2.4 Scop of a Name

- *define:* name are **visible** in scope

global scope

- *visible:* can be used throughout the program

Nested Scopes

```
{
    // ...
    {
        //...
    }
    //...
}
// ...
```

2.3 Compound Types

- *define:* a type that is defined **in terms of another type**

generallay declaration

- **base type** + a list of **declarator**

> declarator: more than vatiable's name

2.3.1 References

> we use the term reference,we mean "lvalue reference"

- *declarator:* **&** variable-name
- *code-look:* `int var = 1024;int &refVar =  var;`
- *wraning:* Because there is no way to rebind a reference, references **must be initialized.**

A Reference Is an Alias

> A reference is not an object.Instead,a reference is just another name for an already existing object.

- *wraning:* Because references are not objects, we may **not define a reference to a reference**.

Reference Definitions

```
int i = 1024,i2 = 2048;
int &r = i,&r2 = i2;
```

- *wraning:* `int &refVar = 10; // error: initializer must be an object`
- *wraning:* `double var = 3.14;int &refVar = var; // error: initalizer must be an int object`

2.3.2 Pointers

- *define:* a compound type that **"points to" another type**

- *wraning:* **pointer is an object**
- *default initialized:* the same to built-in type
- *declarator:* ***** variable-name
- *code-look:* `int *p1,*p2;`

Taking the Address of an Object

- *use:* **&**
- *code-look:* `int var = 1024;int *p = &var;`
- *wraning:* objects have address,so **reference does not has address**
- *wraning:* we may not define a pointer to a refrence

Pointer Value

1. point to an object
2. point to the location just immediately past the end of an object.[2]
3. null pointer
4. invalid

Using a Pointer to Access an Object

- *use:* **dereference operator(*)**
- *code-block*

```
int var = 1024;
int *p = &var;
cout << *p;      // call dereference operator( * )
```

Null Pointers

- *define:* **does not point to any object**
- *code-look:* `int *p = nullptr;// C++ 11 defines nullptr`
- *wraning:* `int *p = 0; // ok`
- *wraning:* `int zero = 0; int *p = zero; // error :zero is an int object but the address of an int object`

Assignment and Pointers

```
int i = 42;
int *pi = 0;
int *p2 = &i;
int *pi3;
pi3 = pi2;
pi2 = 0;
```

- *keep in mind:* assignment changes its **left-hand operand.**

Ohter Pointer Operations

- *use in condition:* if the pointer is **0**,then the condition is **false**,else **true**
- *compare:* two valid pointer of **the same type**,can use (==) and (!=)

void * Pointer

- *define:* hold the address of any object

2.3.3 Understanding Compound Type Declarations

```
int i = 1024,*p = &i,&r = i;
```

Defining Multiple Variables

```
// ①
int *p1,*p2;
// ②
int* p1;
int* p2;
```

- *advice:* choose a style and use it consistently

Pointers to Pointers

```
int var = 1024;
int *pv = &var;
int **ppv = &pv; // pointers to pointers
```

Reference to Pointers

```
int var = 1024;
int *p = nullptr;
int *&r = p;          // reference to pointers
r = &var;
*r = 0;
// result: var == 0
// *&r     : r is a reference,because & is the closest symbol
// int *&r : r is a reference to a pointer to an int
```

- *wraning:* from variable'name,read the definition **right to left**

2.4 const Qualifier

> Because we can't change the value of a const object after we create it, it **must be initialized.**

- *code-look:* `const int buffSize = 512;`

Initialization and const

- *wraning:* `int var = 42;const int cVar = var; // ok`

By default,const Objects Are Local to a File

- *single instace of a const variable:* use **extern** on both its **definition and declaration**
    - code-look

```
extern const int buffSize = fcn(); // file.cpp
extern const int buffSize;         // file.h
```

2.4.1 Reference to const

- *define:* bind a reference to an object of a **const type**

```
const int cVar = 1024;     // ok
const int &rcVar = cVar;   // ok
int &rcVar2 = cVar;        // error
rcVar = 0;                 // error
```

Initialization and References to const

```
int i = 42;
const int &r1 = i;       // ok
const int &r2 = 42 * 42; // ok
int &r3 = 42;            // error

/*  <<
*   what "const int &r2 = 42*42" doing?
*
*   const int temp = 42*42;
*   const int &r2 = temp;         // is bound to a temporary object
*/
```

- *temporary object:* **unnamed object created by the compiler**

A Reference to const May Refer to an Object That Is Not const

```
int i = 42;
const int &ri = i;
ri = 0;      // error
i = 0;       // ok
```

2.4.2 Pointers and const

pointer to const

- *code-look*

```
const double pi = 3.14;
double *ptr = &pi;           // error
const double *cptr = &pi;    // ok
```

- *wraning:* **pointer to const object** and **const pointer to object**

const Pointers

- *code-look*

```
int num = 0;
int *const p = &num;  // p can't change
const int *p2= &num;  // *p can't change
```

2.4.3 Top-Level const

top-level const

- *define:* **const pointer**

low-level const

- define:pointer to a variable,but we can **not use pointer to change the variable**

2.4.4 constexpr and Constant Expressions

constant expression

- *define:* is an **expression** and its **value cannot change** and value **can be evaluated at compile time.** (*three points*)
- *depend:* **type** and **initializer**
- *code-look*

```
int expr = 20;        // expr is not a constant expression.
const int expr2 = 20; // yes
const int expr3 = expr2 + 1; // yes
const int expr4 = get();     // no,compile time can not know
```

constexpr Variables

- *defined standard:* **C++ 11**
- *do what:* ask the compiler to verify that a variable is a constant expression.[3]
- *how:* **Variable** declared as constexpr are **const** and must be **initialized by constant expression.**
- *wraning*

```
constexpr int sz = size();
/*
*   ok only if size is a constexpr function
*/

int num = 20;
const int var = num; // ok,but is not constant expression
```

Literal Types

- *define:* can use constexpr
- *such that:* **arithmetic、reference、pointer type**
- *wraning:* our class、IO library...and so on are not Literal Type

```
constexpr int *p = nullptr; // p only can be initialized by 0 or nullptr or fixed address
(inside the function,the address of variable are not fixed address)
```

Pointer and constexpr

- *wraning:* when we define a pointer in a constexpr declaration,the constexpr specifier applies to the pointer,not the type.[4]
- *code-look*

```
const int *p = nullptr;     // low-level const
constexpr int *q = nullptr; // top-level const
```

- *wraning:* when use constexpr in pointer,it is **top-level const**

2.5 Dealing with Types

2.5.1 Type Aliases

> **Reference** is the **Variable Alias**

- *how to define type aliase*
  - use ***typedef***: `typedef  double wages;// wages is the type alia of double`
  - use ***using***: `using wages = double;// C++ 11`

Pointer、const、and Type Aliase

```
typedef char *pstring;
const pstring cstr = 0; // const pointer,point to char
// if we want const char *,we may be re-typedef
typedef const char *pstring;
pstring cstr = 0;
```

2.5.2 The auto Type Specifier

- *defined standard:* **C++11**
- *wraning:* must be **initialized**
- *wraning:* `auto sz = 0,pi = 3.14; // error`
- *wraning:* `auto sz = 0,num = 3;  // ok`

Compound Types,const、and auto

> The type that the compiler infers for auto is **not always exactly** the same as the initializer's type.

1. **Reference**: The compiler uses that **object's type** for auto's type deduction

```
int i = 0,&r = i;
auto a = r; // auto is int
```

2. **const: ignore top-level const**

```
int i = 0;
const int ci  = i,&cr = ci;
auto b = ci; // ignore top-level const,auto is int

/*
* if you want the deduced type to have a top-level const
*/
const auto f = ci;
```

- *wraning:* ***** and **&** is part of a particular **declarator** and **not part of the base type** for the declaration.

2.5.3 The decltype Type Specofier

- *defined standard:* **C++11**
- *code-look:* `decltype( f() ) sum = x;// the type of sum is the return-type of f()`
- *wraning:*

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0;
decltype(cj) y = x; // decltype(cj) is reference to int.
```

decltype and Reference

```
int i = 42,*p = &i,&r = i;
decltype(r+0) b; // int
decltype(*p) c;  // error  reference must be initialized
```

- *wraning:*
  - get reference type: **decltype((variable))** or **decltype(*p)** (*p is a pointer*) or or **decltype(reference)**
  - get variable type:**decltype(variable's expression)**

2.6 Defining Our Own Data Structures

---

1. 不确定时，最好显式指定***signed char*** 或者 ***unsigned char***↩↩

2. 指向紧邻对象所占空间的下一个位置↩

3. constexpr 类型用于告诉编译器去判定是否变量的值是否是 constant expression↩

4. 当在constexpr中定义了指针，限定符constexpr只对指针有效，与指针指向的对象无关。↩