by Leedehai

Chapter 1 Getting Started with C++

Please refer to the *Introduction* file. □

Chapter 2 Setting Out to C++

2.1 The `main()` function

1. The header of `main()` function could be `int main()` or `int main(void)`. Other fashions, say, returning a `void`, might be allowed by some complier, but are not compatible with the C++ standard.
2. There is a *return statement* `return 0;` at the end of the `main()` function. Nevertheless, you may omit this statement since the compiler adds an implicit return statement at the end if it reaches the end without encountering an explicit one.
3. `main()` function is the only function which does not need or allow a function declaration.
4. Every standalone program needs one and only one `main()`, and it is the starting point of the entire program. However, there are a few exceptions:
   - In Windows programming you can write an *dynamic link library* (DLL) . In DLL there is no need for a `main()` because it is not a standalone program - DLLs are called by other Window programs.
   - Programs designed for some special platforms, say, robot control chips, may not need a `main()`.
   - There might be an hidden `main()` in some programming environment. In this case, the environment provides another explicit starting function (e.g. `_tmain()`). □

Chapter 3 Dealing with Data

3.1 New data type introduced in C++11: `unsigned long long` and `long long`

1. (Supplementary) On the size of integer types
   - A `short` integer is at least 16 bits wide. (Typical: 16 bits)
   - An `int` integer is at least as big as `short`. (Typical: old platforms: 16 bits; now: 32 bits)
   - A `long` integer is at least 32 bits wide and at least as big as `int`. (Typical: 32 bits)
   - A `long long` integer is at least 64 bits wide and at least as big as `long`.
2. `unsigned long long` is as big as `long long`.

3.2 More about types

1. Keyword `unsigned` is equivalent to `unsigned int` when it is used individually.
2. Apart form `bool` and `char`, when `unsigned` is not prefixed to a built-in type notation, then the type is defaulted to `signed`. For example, `short` means `signed short` when used without a prefix `unsigned`.
3. `bool` stands for Boolean type. It has stores zero for `false`, and nonzero for `true`. It has the size of 1 byte.
4. `char` also has 1 byte. It is up to the compiler implementation to decide whether `char` is signed or unsigned. So if you do care about the existence (or absence) of the sign bit, you can use `signed char` and `unsigned char`, respectively.
5. The size of `char` is implementation-dependent. Its size should at least hold the basic character set of the compiler implementation. ASCII or EBCDIC, each of which can be accommodated by 8 bits, are usually designated as the basic character sets, so the typical size of a `char` is 8 bits. Nonetheless, 16-bit or even 32-bit `char` may be used in some implementations.
6. `wchar_t` has the same size and sign properties as its *underlying type*, which is one of the integer type (i.e. `unsigned short`, `(signed) int`, or else). The purpose of introducing `wchar_t` (wide character type) is to hold extended character sets, say, UTF-8 or ISO8859, that are larger than the compiler implementation's basic character set, which is commonly (but not exclusively) ASCII or EBCDIC.
7. `char16_t` has exactly 16 bits and is unsigned. C++ uses prefix `u` for `char16_t` character and string literals, as in `u'C'` , `u"be good"`, and `char16_t ch1 = u'\u00F6';`.
8. `char32_t` has exactly 32 bits and is unsigned. C++ uses prefix `U` for `char32_t` character and string literals, as in `U'C'`, `U"be good"`, and `char32_t ch2 = U'\U0000222B';`.

3.3 About "byte"

There is an intriguing discrepancy between the common definition of "byte" and the definition in C++.

- In the common definition, a *byte* contains 8 bits.
- In the C++ definition, a *byte* contains the minimum number of bits which can hold the compiler implementation's basic character set, say, ASCII. Therefore, though a byte, by this definition, usually contains 8 bits, it is not always the case. A 16-bit or even 32-bit byte might be used.

3.4 Keyword `sizeof`, and header `climits`, `cfloats`

This section discusses the sizes of types.

> The prefix 'c' in the C++ header names `climits` and `cfloat` indicates that these headers are based upon C headers `limits.h` and `float.h`.

- To check the size of one data type, a variable or a literal, use keyword `sizeof`, as in `sizeof(char)`, `sizeof(a)`, and `sizeof('c')`. It returns the size in (C++'s) bytes.The header
- `<climits>` defines symbolic constants to represent various type limits. Values inthis header is provided by the compiler vendor, for different implementations may adoptdifferent sizes of types. *Therefore portability issues may arise*.

  e.g.

Symbol	Meaning
`CHAR_BIT`	Number of bits in a `char`, reflecting the size of byte defined in the implementation. (typical: 8)
`CHAR_MAX`	Maximum `char` value, reflecting whether `char` is signed. (typical: signed8 bits 127)
`USHRT_MAX`	Maximum `unsigned short` value. (typical: 16 bits 65535)
`INT_MIN`	Minimum `int` value. (typical: 32 bits -2 147 483 648)
`INT_MAX`	Maximum `int` value. (typical: 32 bits 2 147 283 648, over 2 billion)
`UINT_MAX`	Maximum `unsigned int` value. (typical: 32 bits, over 4 billion)
`ULLONG_MAX`	Maximum `unsigned long long` value. (typical: 64 bits, nearly 2×10^{19})

- The header `<cfloat>` contains symbolic constants representing information on floating-point types. Similar to `<climits>`, these information is provided by the compiler vendor, and *portability issues may arises*.

  e.g.

Symbol	Meaning
`FLT_MANT_DIG`	Mantissa digits of a `float`. (typical: 24)
`DBL_DIG`	Number of significant digits of a `double`. (typical: 15)
`DBL_MAX`	Maximum `double` value. (typical: nearly 1.8×10^{308})
`FLT_MIN_10_EXP`	Minimum exponent value of a `float`. (typical: -37)

3.5 Numeric literals of integer types

- The compiler treats integers as `int` by default unless one of the two scenarios occur: (i) `int` is too narrow to hold the integer, or (ii) there is a suffix at the end of the literal. Here are the compiler's solutions to such scenarios:
  - (i) `int` is too narrow: the compiler will then try to store the integer in the smallest of the following types:
    - for decimal integer literals (say, `5` , `100`): `int` , `long` , `long long`
    - for octal/hexadecimal integer literals (say, `05` and `0x5` for 5, `0144` and `0x64` for 100): `int` , `unsigned int` , `long` , `unsigned long` , `long long` , `unsigned long long` .
  - (ii) there is a suffix: the suffix denotes the intended type of the literal: `l` , `L` for `long` ; `u` , `U` for `unsigned int` ; `ul` (in any combination of orders and uppercase or lowercase) for `unsigned long` ; `ll` , `LL` for `long long` ; `ull` , `Ull` , `uLL` , `ULL` for `unsigned long long` .

For instance, `long a = 5L;` and `unsigned long b = 5Lu`.

3.6 Numeric literals of floating-points

- There are 2 notations for floating-point literals
  - ordinary notation: `8.0`, `8.`, `.5`, `2.65` (a decimal point is necessary)
  - E notation: `2.2e+8` for 2.2×10^8, `7E1` for 7, `.025e2` for 2.5, `+2.e-1` for 0.2, `-2.E0` for -2

    here are some rules for the notation: (i) the sign `+`/`-` of the significand or the exponent can be omitted; (ii) the decimal point or the fraction part of the significand can also be omitted; (iii) both `e` or `E` to denote the exponent part are valid, and there is no white space before or after it; (iv) the exponent must be an decimal integer.
- The compiler regards a floating-point literal as `double` by default, except the literal is suffixed with `f`, `F` for `float`, or `l`, `L` for `long double`.

3.7 More about floating-points

- The advantage of using floating-points over integers: (i) to represent non-integer numbers (whish is blatantly obvious), and (ii) to represent very large or small numbers, thanks to the wide range of its exponent.
- However, there is some disadvantages: (i) calculations involving floating-points might be slower on most platforms, and (ii) precision of the significand might be compromised. For example, typically, 2 147 483 647 can be precisely stored in an `int`, but in a `float` it becomes 2.14748 $\times10^9$, namely 2 147 480 000.
- `float` guarantees 6-digit precision, and `double` guarantees 15-digit precision.
- Division operation. In expression `a/b`, if and only if both `a` and `b` (variable or literal) are of integer type, then the value of this expression is of integer type. Hence `5/2` yields 2, and `5.0/2` yields 2.5 correctly.

3.8 Automatic (implicit) type conversions

- Situations where automatic type conversions take place: (i) assigning a value of one type to a variable of another type, (ii) evaluating an expression which contains multiple types, and (iii) pass an argument of one type to a function's formal argument of another type.
- Situation 1: assignment (including variable initialization). For expression `A = B`, the value of `B` (be it a variable or a literal) is converted to the type of `A`, and thereafter is assigned to `A`. For instance, `float a=3;`: `3` is initially regarded as an `int`, but it is converted to a `float` before assigned to `a`. So `a` stores a `float`.

  Note: assigning a large value to a smaller type may cause problems.
  - <u>Compromise of precision</u>. For instance, after assigning literal `2111222333`to a `float` variable, the resulting value of that variable is 2.11122×10^9, because `float` guarantees only 6-digit precision (in some compiler implementation the precision might be higher). For another instance, assigning a floating-point to an integer type would cause the fraction part of the value to be truncated (NOT rounded).

- Error of value (a more egregious problem). For instance, assigning literal `98304` (`0x18000`) to a `short` might be troublesome, because the resulting value of that `short` may be -32768 (`0x8000`) - typically a `short` contains only 16 bits. Plus, the value of the expression `short a = 98304` is -32768 instead of 98304.
- Assigning a large number to an unsigned small type: only the lower bits would be stored; assigning a small number to a signed small type: implementation-dependent.

* Situation 2: expression evaluation.
  - *Integral promotion*: the compiler automatically converts `bool`, `char`, `unsigned char`, `signed char`, and `short` to `int`; and if `short` is smaller than (instead of equal to) `int`, then `unsigned short` is converted to `int`, otherwise is converted to `unsigned int`.

    Plus, `wchar_t`, `char16_t`, and `char32_t` are converted to the smallest type which can hold their value, respectively: `int`, `unsigned int`, `long`, `unsigned long`.
  - If the expression contains multiple types, then the rule is: *converts the small to the large*. In details:
    1. If either operand is type `long double`, the other operand is converted to `long double`.
    2. Otherwise, if either operand is `double`, the other operand is converted to `double`.
    3. Otherwise, if either operand is `float`, the other operand is converted to `float`.
    4. Otherwise, the operands are integer types and the integral promotions are made.
    5. In that case, if both operands are signed or if both are unsigned, and one is of lower rank than the other, it is converted to the higher rank.
    6. Otherwise, one operand is signed and one is unsigned. If the unsigned operand is of higher rank than the signed operand, the latter is converted to the type of the unsigned operand.
    7. Otherwise, if the signed type can represent all values of the unsigned type, the unsigned operand is converted to the type of the signed type.
    8. Otherwise, both operands are converted to the unsigned version of the signed type. *Rank of integer types*: `(unsigned) long long` > `(unsigned) long` > `(unsigned) int` > `(unsigned) short` > `(signed/unsigned) char` > `bool`. Plus, `wchar_t`, `char16_t`, and `char32_t` has the same rank as their underlying types, respectively.
* Situation 3: passing arguments to functions. Obvious.

3.9 Forced (explicit) type conversions (type casts)

There are two kinds of type casts: *C-style casts* and *named casts*. Note: Although necessary at times, type casts are inherently not safe.

* *C-style casts*: two patterns -
  - `(typeName) value` : classic C-style. e.g. `(float)2.0`, `(char *) pa`
  - `typeName(value)`: function-call style. e.g. `float(2.0)`, `(char *)(pa)`

- C++ language developer Bjarne Stroustrup found C-style casts too lax - in other words, dangerously unlimited in its possibilities. 4 stricter type casts, namely *named casts*, are introduced, denoted by keywords `static_cast`, `dynamic_cast`, `reinterpret_cast`, and `const_cast`. Format: `named_cast<objectType>(operand)`.
  - `static_cast`: the most basic yet commonly used. It can used to replace C-style casts.

    Features:

    - Casts between built-in types. e.g. `static_cast<float>(2.0)`, `static_cast<unsigned long long>(a)`.
    - Converts null pointer to pointer of a given type. e.g. `static_cast<int *>(0)`, `static_cast<List *>(NULL)`.
    - Converts pointers of any type to `void *`. e.g. `static_cast<void *>(p)`
    - Converts between base class pointers and derived class pointers. Though converting from base class pointers to derived class pointers is permitted, it is dangerous.

      Note:

      (i) it CANNOT convert pointers to another pointer type which is not `void *`;

      ```
      int a = 1;
      int *pa = &a;
      void *pa2 = static_cast<void *>(pa); //ok
      float *pa3 = static_cast<float *>(pa); //error
      // Note: the statement below is also an error. The error occurs
      // when you try to convert 'void *' to 'int *' implicitly.
      int *pa4 = static_cast<void *>(pa); //error
      ```

      (ii) it CANNOT convert pointers of any type to non-pointer types.

      ```
      int b = 1;
      int *pb = &b;
      int npb = (int)pb; //ok. This is the lax C-style.
      int npb2 = static_cast<int>(pb); // error
      ```

      (iii) it CANNOT convert non-pointers to any pointer types.

      ```
      int c = 1;
      int *p = (int *)c; //ok. This is the lax C-style. Dangerous.
      int *p2 = static_cast<int *>c; //error
      void *p3 = static_cast<void *>c; //error
      ```

      (iv) it CANNOT convert reference to another reference types.

```cpp
int d = 1;
int &rd = d;
float &rd2 = (float &)rd; //ok. Meaningless and dangerous.
float &rd3 = static_cast<float &>(rd); //error
// Because rd is an alias of d, so the expressions above
// essentially mean converting a non-reference to a reference, which
// is meaningless.
```

- `dynamic_cast`: Dynamic cast. It will perform runtime safety check (if fails, it will return `NULL`).

  Features:

  - Safely converts between base class pointers and derived class pointers.
  - Virtual function must exists in the base class. In other words, the base class needs to be of polymorphic type.
  - Permits converting between different derived class, provided they are derived from the same base class. But the result is `NULL`.

  ```cpp
  class Base{int x; virtual void virtFunc() {}}; //polymorphic
  class DA : public Base {int yA;};
  class DB : public Base {int yB;};
  int main()
  {
      Base parent; Base *pParent = &parent;
      DA childA;   DA *pChildA = &childA;
      DB childB;   DB *pChildB = &childB;
      // cast from base class pointer to derived class pointer
      DA *pChild_fromBase = dynamic_cast<DA *>(pParent); //ok
     // cast from derived class pointer to base class pointer
      Base *pParent_fromDA = dynamic_cast<Base *>(pChildA); //ok
     // cast from one derived class pointer to another derived
     // class pointer. Result: pChild_fromDB is NULL.
      DA *pChild_fromDB = dynamic_cast<DA *>(pChildB); //ok
      return 0;
  }
  ```

- `reinterpret_cast`: performs a low-level reinterpretation of the bit pattern of its operands. Dangerously yet sometimes usefully, the following behaviors which is prohibited in `static_cast`, is allowed here.

  - Converts pointers to another pointer type.
  - Converts pointer to non-pointer type.
  - Converts non-pointer to pointer type.

```cpp
char ch = 'c';
char *pch = &ch;
int n = 3;
int *pn = reinterpret_cast<int *>(pch);  //ok. 'char *' to 'int *'
char *pn2 = reinterpret_cast<char *>(n); //ok. 'int' to 'char *'
int n2 = reinterpret_cast<int>(pch);     //ok. 'char *' to 'int'
int n3 = reinterpret_cast<int>(ch);      //error. 'char' to 'int'
int n4 = static_cast<int>(ch);           //ok. 'char' to 'int'
```

Note: consider `ip` is a pointer to `int`, and `pc = reinterpret_cast<char *>(ip);` converts it to a pointer to `char`. Any subsequent use of `pc` will assume that the value it holds is a `char *`, and the compiler has no way of knowing that it actually holds a pointer to `int`. Thus problems may arise if you do not take good care of this fact.

For example, if you use `delete pc;` to free the memory, it actually frees one byte (the size of `char`) of memory instead of four bytes (the size of `int`).

- `const_cast`: remove or add the variable's `const` or `volatile` property. However, it is worth noting that removing the `const` property of a `const` object in order to write to it is undefined. The codes below illustrates such scenario:

```cpp
//Valid but NOT intended       --<WRONG WAY>--
const int a = 0;
a = 1;  // error. You cannot assign to a variable that is const.
int *pa = &a; //error. Invalid conversion from 'const int *' to 'int *'
int &ra = a;  //error. Invalid initialization of reference type 'int &'
              //from expression of type 'const int'
int &ra = const_cast<int &>(a);
ra = 1;       //no error.
//The result of the following statements is dependent on the compiler.
std::cout << a << ", " << ra <<std::endl; // 0, 1 -- not 1, 1
std::cout << *(&a) << ", " << *(&ra) <<std::endl; // 0, 1 -- not 1, 1
// This behavior is irregular and unintended. You should avoid it.
```

  - The following code illustrates an appropriate application of `const_cast`:

```cpp
int x = 0;
const int &rx = x; // rx is set to be const inadvertently.
rx = 1;            // not permitted.
int &rrx = const_cast<int &>(rx); // regain write access to x.
rrx = 1;           // ok
//The result of the following statement is 1, 1, 1
std::cout << x << ", " << rx << ", " << rrx << endl;
/*
the following code does the same thing:
const int *px = &x; // px is set to be const inadvertently.
*px = 1;            // not permitted.
int *ppx = const_cast<int *>(px);
*ppx = 1;           //ok
*/
```

> Q: In the code snippet above, why don't you modify the declaration of `rx` (a.k.a. delete the prefix `const`)?
>
> A: In some large projects, the declaration of `x` and `rx` (inadvertently set to `const`) might be in another file (for instance, third-party code module), hence it is inconvenient and error-prone to mess around the original declaration. Or, maybe we would like to use `const` to prevent unintended modification to `rx` (essentially `x`), but temporary necessity to override the write-protection arises.

- Another suitable context of applying `const_cast` involves overloaded functions. Consider a function `shorterString`:

```cpp
// return a reference to the shorter of two strings
const string &shorterString(const string &s1, const string s2)
{
  return s1.size() <= s2.size() ? s1 : s2;
}
```

The function takes and returns references to `const string`. What if we want to have another version of `shorterString` that takes and returns non-`const` plain references? We can overload this function:

```cpp
string &shorterString(string &s1, string s2)
{
  return s1.size() <= s2.size() ? s1 : s2;
}
```

Using `const_cast`, we can overload the function in a more convenient way - without rewriting the code. This convenience is more salient if the original function's implementation is long and complicated, and thus provides relief when you realize you have to modify the code (modifying similar codes in multiple places is drudgery):

```
string &shorterString(string s1, string s2) // better reusability
{
  // with const_cast, we can call the function's original version
  auto &r = shorterString(const_cast<const string &>(s1),
                          const_cast<const string &>(s2)); // add const
  return const_cast<string &>(r); // remove const
}
```

- *Note*:

  (i) The object type and the type of the operand of `const_cast` must be the same, other than the existence or absence of `const` or `volatile`. Plus, casting from `const` to `volatile` or vise versa, casting from `volatile const` to `const` / `volatile` or vise versa, are also permitted:

  ```
  int n = 0;
  const int *pnc = &n;
  volatile int *pnv = &n;
  // the following are valid:
  volatile const int *pnvc1 = const_cast<volatile const int *>(pnc);
  volatile const int *pnvc2 = const_cast<volatile const int *>(pnv);
  volatile int *pnv2 = const_cast<volatile int *>(pnc);
  const int *pnc2 = const_cast<const int *>(pnv);
  int *pn1 = const_cast<int *>(pnc);
  int *pn2 = const_cast<int *>(pnv);
  ```

  (ii) The object type in a const_cast must be a pointer, reference, or pointer to member to an object type:

  ```
  int d = 0;
  volatile int &rd = d;
  int rrd = const_cast<int &>(rd); // ok

  volatile int e = 0;
  int ee = const_cast<int>(e); // error
  ```

3.10 Initialization: combines assignment with declaration

- Initialization syntaxes:
  - traditional syntax coming from C:

    non-array: `type varName = initializer;` or `type varName = {initializer};`

    array: `type varName[size] = {initializerList};`
  - new C++ style:

    non-array: `type varName(initializer);` or `type varName{initializer};`

    array: `type varName[size]{initializerList};`

```
//traditional:
int a1 = 1;
int a2 = {1};
int a3 = {}; // a3 is set to 0
int b[2] = {0, 1};
//new:
int c1(1);
int c2{1};
int c3{}; // c3 is set to 0
int d[2]{0, 1};  // array initializer can only be enclosed by braces.

// combining () and {} is not permitted. Because {0,1} is not itself
// an initializer - rather, it contains two initializaer 0 and 1.
int f[2]({0,1}); // WRONG
```

- C++ added the parentheses form `()` of initialization to make initializing ordinary variables more like initializing class variables.
- Furthermore, C++11 makes it possible to use the braces syntax (`{}` with or without the `=`) with all types - a universal initialization syntax. This form of initialization is called *list-initialization*. Maybe this new syntax would replace the former ones, relegating them to historical oddities.
- In list-initialization, C++ protects against type narrowing. For instance, floating-point types cannot be converted to integer types, because the compiler knows such conversion has the risk of compromising the fraction part; integer types can be converted to `char` only if the compiler can tell `char` is able to hold the value.

```
char ch1 = 2000; // allowed. Though narrowing occurs here.
char ch2 {2000}; // error. 'char' cannot hold 2000
char ch3 {255};  // allowed. 255 can fit in a 'char'
int n1 = {2.1};  // error.
int n2 {2.0};    // still error. Though the fraction part is 0.
```

☐