Chapter 4 Compound Types

4.1 Creating and using arrays (1-dimensional)

- A *array* is a data form than can hold multiple values, all of one type (*homogeneity*). Array is not a data type on its own, but rather it is a "derived" form of data type, much like pointers. Declaration: `type arrayName[size];`
- A *subscript*, a.k.a. *index*, denotes elements in an array. **Subscripts start with 0.**
- Array Initialization: there are several ways to initialize an array.

```
int power[5]; // no initialization. ok
int money[5] = {1,2,3,4,5}; // ok. traditional C syntax
int greed[5]{1, 2, 3, 4, 5};// ok. new C++ syntax
```

- It is permitted to partially initialize an array, the rest of which are set to 0 automatically.

```
int art[5] = {1, 2}; // <=> int art[5] = {1, 2, 0, 0, 0};
int music[5]{1, 2};  // '=' sign van be dropped, as covered in 3.10
int drama[5] = {0};  // a useful way to set all the elements to 0
int anotherDrama[5]{0}; // set all to 0
int dance[5] = {};   // new C++11 way to set all the elements to 0
int anotherDance[5]{};  // set all to 0
```

- It is also permitted to initialize an array without designating its size (and therefore the bound) explicitly. Leaving the brackets `[]` empty lets the compiler count the size for you.

```
int beauty[] = {1,2};   // the size is 2
int anotherBeauty{1,2}; // the same
int size = sizeof(beauty) / sizeof(int); // returns the size, so the program
                                         // knows how large the array is.
int spoiler[] = {};     // invalid, of course. Unspecified bound.
```

- List-initialization (`{}`) protects against narrowing. As mentioned in 3.10.

```
long boys[] = {1, 2, 3.0}; // error. 3.0 is 'double', too big for 'long'
char girls{'h', 255, 512}; // error. 512 is out of the range of 'char'
int babies[2];
babies[1] = 2.0;           // allowed. It does not involve list-
                    // initialization. In fact, it is not an
                           // initialization at all; it is just an
                           // ordinary assignment.
```

- The label (a.k.a., name) of an array is loosely regarded as the name of the pointer point to the array's first element, and the pointer is of the same type as the array. Plus, if you apply `&` to the array name, you will get the address of the array `a`, which is the same value as pointer `a`.

```
int a[5] = {-1, 5, 2, 2, 0}; // a is a pointer of 'int *'
std::cout << a << ", " << &a << ", " << a+1 << ", " << a[0] << ", "
          << *(a+1) << std::endl;
// A possible output:
// 0117F99C, 0117F99C, 0117F9A0, -1, 5
// Note: a brief reminder on pointer addition:
//   because 'a' is of type 'int *', so a+1 essentially
//   means (void *)a + sizeof(int).
```

- Note: the array name `a` is NOT a real pointer - it is, after all, a name of an array. For instance, `a++` or `a = a + 1` generates an error, because you cannot assign a value to a name (a name has no value). Conceptually, the name of an array is an rvalue, which appears on the right side of assignment operator `=`, but is not an lvalue, which appear on the left side. A real pointer, like other ordinary variables, is both an rvalue and an lvalue. Plus, `sizeof(a)` returns the size, in bytes, of the whole array, instead of the size of a pointer.
- Assume `a` is an array. We already know from above that `a` is, loosely speaking, a pointer point to the array's first element. However, `a` is NOT a pointer point to the array itself, which is somewhat counterintuitive. The address of the whole array is `&a`. `&a` and pointer `a` has the same value - the address of the array's first element's first byte, but they are conceptually different: `&a` is the address of the whole block of memory that holds the array, whereas `a` just stores the address of the block of memory array's first element occupies.

  This discrepancy can be showed in this example:

```
int a[17];
std::cout << a << ", " << a+1 << std::endl;
// output: 00D5FE8C, 00D5FE90
// 0x00D5FE90 - 0x00D5FE8C = 0x4 = 4 (the size of 'int')
std::cout << &a << ", " << &a+1 << std::endl;
// output: 00D5FE8C, 00D5FED0
// 0x00D5FED0 - 0x00D5FE8C = 0x44 = 68 = 17 * 4
```

  It shows that the value of `a+1` is the value of `a` plus `1 * sizeof(int)`, whereas the value of `&a+1` is the value of `&a` (equal to the value of `a`) plus `17 * sizeof(int)`, which is the size of the whole array. This example suggests that `a`'s value is regarded as the address of a `int`, yet `&a` is treated as the address of the whole array.

  - Another way to put this is that `a` is a "pointer-to-int", or `int *`, and `&a` is a "pointer-to-array-of-17-`int`s", or `int (*)[17]`. You can actually assign `&a` to a real pointer: `int (*ptr)[17] = &a;`; but `int (*ptr)[17] = a;` generates an error.

- The C++ *Standard Template Library* (STL) provides an alternative to arrays called the `vector` template class in header `<vector>`, and C++11 STL adds an `array` template class in header `<array>`. These alternatives are more sophisticated and flexible than the built-in array composite type.

4.2 C-style strings

A *string* is a series of characters stored in consecutive bytes of memory. C++ has two ways of dealing with strings: firstly, *C-style* string taken from C, and secondly, `string` class.

- The last character of every string is *null character* `\0`. There are two ways of initializing a C-style string:

  (i) using list-initializations, as in other arrays.

```
char happy[6] = {'h', 'a', 'p', 'p', 'y', '\0'}; // a string
char smile[7] = {'s', 'm', 'i', 'l', 'e', '\0', '\0'}; // another string
char angry[5] = {'a', 'n', 'g', 'r', 'y'}; // not a string -- no '\0'
```

  (ii) using double-quoted strings (*string literal*s). The compiler appends `\0` at the end.

```
char happy[6] = "happy";   // automatically fills '\0' at the end
char humor[100] = "humor"; // the compiler fills 95 '\0's
char smile[] = "smile";    // let the compiler count. size = 5+1 = 6
char *joy = "joy";         // also permirtted. Let the compiler count.
char love[4] = "love";     // error. No room for any '\0'.
```

- C++ mandates that compilers treat string literals as constants, leading to a runtime error if you try to write new data over them. But not every compiler implementation honor this rule.
- Some compilers use just one copy of a string literal to represent all occurrences of that literal in the program code.
- About `cout` and character arrays: the `cout` object assumes that the variable of `char *` is the address of a string. So if it is given an address of a character array, it will print out its string content - not the array address itself.

```cpp
int integer[5] = {1, 2, 3, 4, 5};
std::cout << integer << ", " << integer+1 << std::endl;
//you may get: 0117F99C, 0117FRA0

char char1[33] = { 'r', 'o', 's', 'e', '\0' };
char char2[] = "violet";
std::cout << char1 << ", " << char1 + 1 << std::endl
          << char2 << ", " << char2 + 2 << std::endl
          << *char1 << ", " << &char1 << std::endl
          << *char1+1 << ", " << &char1+1 << std::endl;
/*you may get:
rose, ose
violet, olet
r, 00EFF9C4
115, 00EFF9E5
*/
// Question: why is (&char1+1) - (&char1) = 0x21 (33)?
// Answer: &char1 is the address of the whole array, so &char+1 means
//         &char1 plus the size of the whole array. &char1 and char1
//         stores the same address value, but their meaning is not
//         the same. Discussed in depth in 4.1.
```

- It is noted in Stanley Lippman's *C++ Primer* (fifth edition) that,

  > WARNING: Although C++ supports C-style strings, they should not be used by C++ programs. C-style string are a surprisingly rich source of bugs and are the root cause of many security problems. They are also harder to use!

  And it is highly recommended in the book that,

  > TIP: For most applications, in addition to being safer, it is also more efficient to use library `string`s rather than C-style strings.

4.3 The `string` class

- The `string` class is simpler to use than the array and also provides a truer representation of a string as a "type". It hides the array nature of a string and let you treat it as an ordinary variable.
- The `string` class is defined in the header `<string>`, and is part of the `std` namespace (as are other standard C++ classes).
- The initialization of a `string` object is the as that of a C-style array, as is demonstrated below.

```
// C-style string
char kinderdarten; // no initialization
char primarySchool = "Mapple Moutain Primary School";
char juniorSchool = {" Nanya Boarding School"}; // since C++11
char highSchool {"Yali Middle School"};         // since C++11
// string object
string gaokao;     // no initialization
string dreamCollege = "Tsinghua University";
string endUpCollege = {"Zhejiang University"};  // since C++11
string studyAbroad {"Stanford Univeristy"};     // since C++11
```

- The `string` class is more than a container of strings. It provided a range of useful methods. For example
  - direct assignment, just like a ordinary variable. It holds true for objects of other classes.

```
char char1[5] = "ab", char2[5];
string str = "cd", str2;
char2 = char1; // invalid. array type 'char [5]' is not assignable
str2 = str1;   // ok.
```

- appending a string to another: `+` and `+=`. For example,

```
string str1 = "how are", str2 = " you?";
string str3 = str1 + str2; // str3 becomes "how are you?"
str1 += str2;              // str1 becomes "how are you?"
string str4 = "Alice, " + str1; // "Alice, how are you?"
```

- other tricks. Such as
  - `at()` (returning the element at a given position),
  - `size()` (returning the string's size, excluding the trailing `\0`),
  - `front()` and `back()` (returning the first / last character), and
  - `append()` (for appending).

The prototypes of these four are:

```
char &std::string::at(size_t);
size_t std::sting::size() const;
char &std::string::front();
char &std::string::back();
// size_t: alias of one of the built-in unsigned integer types,
// defined in some headers.
```

- Other forms of string literals: C++ uses prefixes for string literals of special types other than `char`: `L` for `wchar_t`, `u` for `char16_t` (UTF-16), and `U` for `char32_t` (UTF-32). Plus, C++11 added prefix `u8` to denote UTF-8 strings. For instance,

```
wchar_t title[] = L"President";
char16_t name[] = u"Barack Obama";
char32_t address[] = U"The White House, 1600 Pensylvannia Avenue, D.C.";
char party[] = u8"The Democratic Party";
```

> UTF-8: An encoding scheme based on Unicode. It uses 1 - 4 bytes to represent a character, depending on the numeric value of characters' codes.

Besides, C++11 adds a new kind of string - *raw string*. In a raw string, characters simply stand for themselves - for example, `\n` is not interpreted as the newline character; instead, it is just two ordinary characters: backslash `\` and lowercase letter `n`. The prefix for it is `R`, and use parenthesis `()` as the delimiters inside the double quotes.

```
// Jim "King" Tutt uses "\n" instead of endl.
// using raw string:
std::cout << R"(Jim "King" Tutt uses "\n" instead of endl.)" <<std::endl;
// using ordinary string:
std::cout << "Jim \"King\" Tutt uses \"\\n\" instead of endl." <<std::endl;
```

And if you want to display `()` itself,

```
// "(Who wouldn't it?)", she whispered.
std::cout << R"+*(Who wouldn't?)", she whispered.)+*";
// in short, in this case, the default delimiters () have been repalce
// with +*( and )+*.
```

Lastly, the prefix `R` can be combined with other string prefixes in any combination to produce raw strings of `wchar_t`, `char16_t` and `char32_t`. Like,

```
string str = LR"+*(() is a pair of parenthesis.)+*" << endl;
std::cout << Ru"([] is a pair of brackets)";
char chars[] = UR"({} is a pair of braces)";
```

4.4 More about string I/O

- `cin >>` stops reading input after encountering a *white space*, be it a space, a tab, or a newline. It leaves the white space in the input queue, and append `\0` (or multiple `\0`s) at the end of the string. By the way, we can call `cin.ignore();` to ignore any white spaces at the beginning of the input queue.
- To avoid inconvenience caused by this (we might want to read in a sentence, which may contain spaces), `getline` is introduced. It stops reading at the first newline it meets, and cast the newline away form the input queue. Its usage is demonstrated below:

```
// for C-style strings
char name[20];
cin.getline(name,20);
// for string objects
string address[20];
getline(cin, address);
```

4.5 Structures

A *structure* is a user-definable type, containing one or more member elements (*field*s). Homogeneity of its elements' types is not required.

- Basic usage:

  ```
  // structure definition
  struct car
  {
    string manufacturer; // separated with semicolons
    float priceInDollars;
    int yearOfMaking;
  };

  // use this type to declare objects
  struct car myOldCar; // C-style way
  car myNewCar;        // "struct" can be dropped
  ```

  like any ordinary types, structures can be declared outside or inside a function. It have to be declared before invoking either way.

- About its size and memory alignment

  It is natural to wonder whether the size of a structure type is the sum of the sizes of its members. The answer is NO. The fact is that the size of a structure type is AT LEAST the sum of the sizes of its members, as you can verify with `sizeof`. The mechanism behind this counter-intuitive is call *memory alignment*.

  > Memory alignment:
  >
  > For CPUs, it is more efficient to retrieve bits in chunks of words than in individual bits - because CPU is restricted to accessing memory at the granularity. For instance, a 32-bit CPU reads in instructions or data from peripheral devices 32 bits (a word) at a time, and a 64-bit CPU 64 bits at a time. To accommodate such behavior pattern of CPUs, the compiler compiles the program such that the size of a structure has one or more times the size of the CPU's word size, while has enough spaces for its members.
  >
  > Compiler has different preset *alignment modulo* (8, 16, 32, 64, ... the corresponding word size) for each different object platforms. Programmers may alter the preset modulo with preprocessor directive '#pragma pack(m)', where m is the customized alignment modulo.

- Initialization:

```
// old way
car myCar1 = {"Mercedes-Benz", 55000, 2016}; // list-initialization.
car myCar2 =
{
  "Volkswagon",
  30000,
  2015          // the last term does NOt end with a comma, because the
                // syntax complies with list-initialization.
};
// new C++ way: '=' is optional
car myCar3 {"Mercedes-Benz", 50000, 2009};
car myNextCar {}; // empty braces set all the fields to 0.
```

- More about definition, initialization, and assignment.
  - You can perfectly assign one structure to another, provided they are of the same type. It is called *memberwise assignment*.
  - You can set a member field to a particular value in the structure definition. But variables of that type cannot be initialized.

```
struct data{
  double speedOfLight = 2.99792458E30; // preset
  int blabla;
};
data myData = {3.0E30, 2}; //error
// However:
myData.speedofLight = 3.0E30; // allowed. It is not initialization.
myData.blabla = 2;            // allowed.
```

- You can define a structure type with no name, a.k.a., *anonymous structure*, but you have to declare variables of this type simultaneously (otherwise you cannot refer to this type -- it has no name!).

```
struct {
  std::string motto;
} aryaStark;          // easter egg: Arya Stark said to her one-time
                      // mentor: "A girl has no name".
                      //   -- TV series "Game of Thornes", Season 6
```

- You can combine the definition of a structure and the creation of (one or more) variables of that structure, as is provided in C. You can even initialize the variable you just created.

```
struct integer{
  int a;
} firstNum, secondNum;
```

```
struct fraction{
  float b;
} myFraction = {2.5};
```

```
struct ratio{
  float nominator, denominator;
} firstRatio = {
  2,3
},
secondRatio = {
  1,2
};
```

- Bit Fields in structures: C++, like C, enables you to specify structure members that occupy a particular number of bits. Such fields are termed *bit field*s. This can be handy for creating special data structure that corresponds, say, to a register on a chip.

  The field type can be an integral or enumeration type, and a colon followed by a number indicates the actual number of bits to be used. And you can use unnamed fields to provided spacing.

```
//Bit fields -- a feature suitable for low-level programming
struct register_WatchDogControl { // WatchDog register on some DSP chip
  bool WDFLAG : 1;  // flag of the WatchDog
  unsigned int : 1; // 1 bit unused. Type 'unsigned int' is irrelevant.
  bool WDDIS : 1;   // WatchDog disable toggle
  int WDCHK : 5;    // WatchDog checking field
  int WDCNTR : 8;   // WatchDog counter
};

// can be accessed in the usual manner
register_WatchDogControl regWDCR = {true, 0, 0xF5, 0};
/*
  1 / x / 0 / 1 0 1 0 1 / 0 0 0 0 0 0 0 0
  regWDCR.WDFLAG is 1;
  (1 bit unused)
  regWDCR.WDDIS is 0;
  regWDCR.WDCHK is 1 0 1 0 1 (0xF5 = 1111 0101)
  regWDCR.WDCTR is 0 0 0 0 0 0 0 0
*/
...
regWDCR.WDDIS = true;  // regWDDIS becomes 1
refWDCR.WDCHK = 0xFA ; // regWDCHK becomes 1 1 0 1 0 (0xF1 = 1111 1010)
...
```

- Array of structures.

```
car myCars[4];
car myFirendsCars[2] = {
  {"Volkswagon", 220000, 2002},
  {"Volkswagon", 500500, 2000}
};
myFriendsCars[0].priceInDollars = 260000;
*(myFriendsCars+1).priceInDollars = 501500;
```

4.6 Unions

A *union* is a user-defined data format that can hold different data type but only one type at a time. That is, whereas a structure can hold, say, an `int` and a `long` and a `double`, a union can hold an `int` or a `long` or a `double`. The syntax is like that for a structure, but the meaning is different.

- Defining, declaring and using a union type.

```
union one4all{
  int n;
  long ln;
  double db;
} a;
a.n = 1;   // stores an 'int'
std::cout << a.n << endl;
a.ln = 2;  // stores a 'long', the 'int' value is lost
std::cout << a.ln << endl;
a.db = 3;  // stores a 'double', the 'long' value is lost
std::cout << a.db << endl;
```

- The size of a `union` is at least the size of its largest member. Why is it "at least" instead of "equal to"? The reason lies with memory alignment, as discussed in 4.5.
- You can define a union inside the definition of a structure.

```
struct S{
  int n;
  union {
    int a;
    int b;
  };
};
S mySt;
mySt.n = 1;
mySt.a = 2;
mySt.b = 3; // then mySt.a value is lost
```

```
union U{
  int n;
  struct S{
     int a;
     int b;
   };
};
U myUn;
myUn.n = 1;
myUn.a = 2; // error. 'a' is NOT a member of myUn.
```

The inside union has declared no variable name. If it does, then we must access their members through its intermediate variable name, instead of doing so directly as demonstrated above. An inside structure needs an intermediate variable name.

```
struct S{
  int n;
  union {
    int a;
    int b;
  }insideUnion;
};
S mySt;
myUn.n = 1;
myUn.a = 2;   // error
myUn.insideUnion.a = 2;
myUn.insideUnion.b = 3; // then myUn.insideUnion.a value is lost
```

```
union U{
  int n;
  struct S{  // the name S is optional, but it cannot be used outside
    int a;
    int b;
  }insideStruct;
  S anotherStruct;
};
U myUn;
myUn.n = 1;
myUn.insideStruct = { 2, 3 }; // then myUn.n value is lost.
myUn.anotherStruct = { 3, 2 }; // then myUn.insideSruct value is lost.
```

- Unions often, though not always, are used to save memory space, especially for embedded systems (such as a toaster or a satellite), where space may be at a premium.

4.7 Enumerations

The C++ `enum` facility provides an alternative to `#define` and `const` for creating symbolic constants. It also lets you define new types but in a fairly restricted fashion.

- Basic usage

```
enum spectrum {red, orange, yellow, green, blue, indigo, violet};
spectrum myBand; // declare a variable of this type
spectrum myBand = blue;   // valid

myBand = 2;       // error. Implicit conversion to spectrum not allowed
myBand = spectrum(2);    // allowed.
myBand = (spectrum)2;    // allowed.
```

Here `spectrum` is the name of this enumeration, and `red`, `orange` and so forth are termed `enumerator`s.

  - By default, enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second enumerator, and so on. You can override the preset by explicitly assigning integer numbers, partially or wholly.
    - `enum letter{a = 1, b = 2, c = 4, d = 8, e = 16, f = 32};`
    - `enum letter{a, b = 100, c, d, e = 200, f};` In this case, `a` is assigned `0` by default, and subsequent unspecified enumerators are larger by 1 than their immediate predecessors, hence `c` is 101, `d` is 102, `f` is 201.
    - `enum letter{a, b = 0, c, d = 0};` You can create more than one enumerator with the same value, and the enumerator-values are NOT necessarily incremental. In this case, `a` is 0 by default, `b` is 0, `c` is 1, `d` is 0 again.
    - Enumerators are integers. They can be `int` or types that promote to `int` (such as `char`, `bool` and `short`), plus `long` and even `long long`.
    - The default underlying type for enumerators are implementation-dependent (except for scoped enumerations discussed below). You can customize your choice, with the syntax:

```
enum resident{apartment, house}; // default enumerator type
enum : short swimmingPool {indoor, outdoor}; // user-specified
```

  - What if you assign an invalid value to an enumeration variable? The result is undefined and is dependent on the compiler.

```
myBand = 100; // error. Implicit conversion to spectrum not allowed
myBand = spectrum(100); // undefined
```

    - It is a bit tricky in determining what an "valid" value is. Apart from enumerator values, values falling in the *range* of the enumeration type are also considered valid. Range here is defined as follows:

> "First, to find the upper limit, you take the largest enumerator value.Then you find the smallest power of two greater than this largest value and subtract one; the result is the upper end of the range.
>
> Next, to find the lower limit, you find the smallest enumerator value. If it is 0 or greater, the lower limit for the range is 0. If the smallest enumerator is negative, you use the same approach as for finding the upper limit but toss in a minus sign."
>
> To sum up,
>
> UPPER LIMIT = 2^k-1, with minimum possible positive exponent k, and greater than the largest value;
>
> LOWER LIMIT = min{0, -(2^t-1)}, with minimum possible positive exponent t, and greater than the smallest value's absolute value.

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
bits myFlag;
myFlag = 1;       // INVALID. Interger cannot be
                  // converted implicitly.
myFlag = bits(1); // valid. 1 is an valid value.
myFlag = bits(6); // valid. 6 falls in the range, hence is valid.
```

- Arithmetic operators such as '+' and '++' are not defined for enumerators. Enumerators, however, can be promoted to integers during assignment and expression evaluation.

```
myBand++;                // error. '++' not defined for enumerators
int myColor = blue;      // valid. Integral promotion promoted
                         // enumerators to integers.
color = red + 3;         // valid. red is promoted to integer 0.
myBand = orange + red;   // error. Though orange and red are promoted
                         // integers, integers cannot be converted to
                         // enumerators implicitly, thus cannot be
                         // assigned to an enumeration variable.
int color = orange + red;// valid.
```

- C++11 extends enumerations with a form called *scoped enumeration*. The logic behind this is that with a scope set for each enumeration type, enumerators' identifiers will no longer conflict. For example, in traditional enumeration, the code below generates an error because of name conflicts:

```
// error. Names conflict.
enum shirts{small, medium, large, jumbo};
enum pants {small, medium, large, gigantic};
```

In C++11, a new form of enumeration, namely *scoped enumeration*, solves this problem by having class or structure scope for its enumerators:

```
enum class shirts{small, medium, large, jumbo};
enum class pants {small, medium, large, gigantic};
// or
enum struct shirts{small, medium, large, jumbo};
enum struct pants {small, medium, large, gigantic};
```

When you use the enumerator, you need to qualify them with the enumeration name - telling the compiler which scope you are using.

```
shirts myShirt = shirts::medium;
pants myPant = pants::medium;
shirts anotherShirt = large;      // error, which 'large'?
```

  - Regular enumerators can be promoted to integer type in some situations, such as assignment to an `int` or being used in an expression. But scoped enumerations CANNOT.
  - The underlying type for regular enumerators are implementation-dependent. However, for scoped enumerators, the underlying type is `int`. Plus, you can of course customize your choice, like regular enumerations:

```
enum class salad {small, medium, large}; // type is int by default
enum class : short juice {small, medium, large}; // type is short
```

- Despite the restrictions set forth above, not all compilers honor these provisions. Some, such as Microsoft VC++'s compiler, may allow a more loose stipulation. That being said, you are recommended to adhere to the standard for portability.

4.8 Managing dynamic memory with `new` and `delete`

It is an often-used technique to allocate *unnamed memory* during runtime to hold values, in which case, by the way, pointers are the only access to that memory. In C, you can use library function `malloc()` to allocate memory and `free()` to deallocate them; and you can still do so in C++, provided the header `<cstdlib>` is included.

> Prototypes of `malloc()` and `free()`:
>
> `malloc()`: `void *malloc(size_t size);`
>
> `free()`: `void free(void *ptr);`

C++ introduced new operators `new` and `delete` as alternatives to functions `malloc()` and `free()`, respectively.

- Basic usage: `type *ptrName = new type;` and `delete ptrName;`

```
int *pn = new int;   // allocate a block of memory to store an 'int'
char *pc = new char {'a'}; // list-initialization allowed
...
delete pn;           // deallocate that block of memory
delete pc;
```

- It is possible that a computer might not have sufficient memory available to satisfy a `new` request, especially when the requested block of memory is fairly large. C++ provides tools to detect to detect and respond to such failures. When that is the case, `new` normally responds by throwing an exception. In older implementations `new` returns the value 0. A pointer with the value 0 is called the *null pointer*, and C++ guarantees that a null pointer never points to valid data.
- You should NOT free a block of memory more than once. The C++ standard says the result of such attempt is undefined. Also, you cannot use `delete` to free memory created by declaring ordinary variables.
- `delete pointer;` does NOT mean that the pointer itself is deleted; rather it means to block of memory the pointer once pointed to is deleted, and the pointer itself can be reused.
  - After `delete pointer;`, unless `pointer` is directed to point to another block of memory, attempts trying to access the memory `pointer` once point to might generate a runtime error: the address value `pointer` holds might be changed to an unspecified value (such pointer is termed *unbound pointer*, or *wild pointer*). This is the result of my test on Microsoft VC++:

    ```
    int *pn = new int;
    *pn = 2;
    std::cout << pn << ", " << *pn << std::endl; // output: 00AFEB48, 2
    delete pn;
    std::cout << pn << " " << std::endl;         // output: 00008123
    std::cout << *pn << std::endl;    // the system might crash
    ```

    So it is recommended to set the pointer's address value to `NULL` after `delete`, lest the pointer runs amok.
    - Plus, it is safe to apply `delete` to the null pointer - nothing happens.
- You should always **balance** a use of `new` with a use of `delete`; otherwise, you can wind up with a *memory leak* - that is, memory that has been allocated but can longer used. If a memory leak grows too large, it exhausts all the memory available.

> Memory leak is a kind of hard-to-detect bug, for C++ does not provide any built-in tool to detect them. Worse, C++ does not provide for automatic memory deallocation, also figuratively termed *garbage collection*. Therefore, a program with memory leak might run for a fairly long time (maybe years) before the flaw is exposed and does real damage.

- It is allowed to have two pointers pointed to the same block of memory. But it is recommended that you avoid such deed, for fear that you may mistakenly try to delete the same block of memory twice. And after `delete`, this block of memory is immediately returned to the system, and attempts to access the memory through the other pointer might generate a runtime error.

```
// NOT recommended, but valid
int *pa = new int;
int *pb = pa;
*pb = 2;            // equivalent to *pa = 2;
delete pq;
*pb = *pb + 1;      // the system might crash
```

4.9 Using `new` / `delete` to manage dynamic arrays

- If all a program needs is a single value, you may as well declare a variable in the common way, because that is simpler, if less impressive, than using `new` and a pointer to manage a small data object. More typically, `new` is employed to deal with larger chunk of data, particularly when the size of the chunk is not known until runtime. Allocating the array during compile time is termed *early binding* or *static binding*, meaning the size of the array is built in the program code; whereas with `new`, it is permitted to create arrays and specify their sizes during runtime, and this is called *late binding* or *dynamic binding*.
- Dynamic memory allocation: `type *ptr = new type[size];`

  > The `new` returns the address of the array's 'first element's first byte, to the pointer `ptr`.

```
int *p = new int[2]; int *q = new int[2] {1,2};  // size must be specified
```

  > Note that the brackets `[]` is behind the type name.

- Dynamic memory deallocation: `delete[]ptr;`

  > The presence of the brackets `[]` tells the compiler that your intention is to free the whole array, not just the element pointed to by the pointer. As discussed in 4.1, the name of an array stores the address of the first element, NOT conceptually the address of the whole array. `delete` and `delete[]` can be viewed as different operators.

```
delete[]p;   // free the whole block of memory
delete p;    // just the memory block occupied by the first memory is
          // freed.
```

  > TIP: If your `new` statement has a pair of `[]`, then your corresponding `delete` should also has a pair of `[]` to balance it out.
  >
  > Note that the brackets `[]` is between the keyword `delete` and the pointer.

- Using a dynamic arrays: note the fact that if `a` is a pointer, then `a[n]` is equivalent to `*(a+n)`.

```
double *t = new double {0}, *p = new double[3];
std::cout << "*t is " << *t << ", address = " << t << endl;
delete t;
p[0] = 0; p[1] = 8; p[2] = 8;
std::cout << "p[2] is " << p[2] << ", address = " << p+2 << endl;
delete[]p;
```

- A side note: suppose `a` is an array name. Though `a` can be used to represent the array's address, `a` is an array name after all. That means this statement generates an error: `a = a + 1;`, because you cannot change the "value" of an array name (a name has no value). Discussed in 4.1.
- However, in dynamic memory allocation, `int *a = new int[10];` means `a` is a pointer; though we can treat it as an array name in statement like `a[0] = 1;`, it is essentially a pointer, enabling you can change its value: `a = a + 1;` and this operation complies with pointer arithmetic operation rules.

> The **near** equivalence of pointers and array names is arguably one of the beauties of C and C++, yet it is also controversially one of the flaws.

- Dynamic multidimensional array: a bit more complicated and care-demanding.
  - A multidimensional array consists of multiple subarrays. For instance, declaring a multidimensional array `int a[3][4];` means `a` is an array consisting of 3 elements, and each of the element of `a` is also an array, which holds 4 elements. Therefore, `a`, together with `a[0]`, `a[1]`, `a[2]`, is a pointer. This fact is exemplified in the following code:

```
int a[3][4];
std::cout << a << endl;         // 008FF990, an 'int **' pointer
std::cout << a[0] << endl;      // 008FF990, an 'int *' pointer
std::cout << a[1] << endl;      // 008FF9A0, an 'int *' pointer
std::cout << a[2] << endl;      // 008FF9B0, an 'int *' pointer
std::cout << a[2][3] << endl;   // -858993460 (garbage), a 'int'
```

> NOTE: `a` is a pointer-to-pointer-to-`int` - sounds awkward? `a[0]` is a pointer-to-`int`, hence `a[0]` is of type `int *`. And pointer `a` points to array `a`'s first element, which is `a[0]` (a pointer `int *`), so `a` is `int **`.

Likewise, in a 3-dimensional array `b[3][4][5]`, the following expressions are pointers: `b`, `b[0]`...`b[2]`, `b[0][0]`...`b[0][3]`, `b[1][0]`...`b[1][3]`, `b[2][0]`...`b[2][3]`. And expressions `b[0][0][0]`...`b[2][3][4]` are integers.

> TIP: for an n-dimensional array `a` of type `type`:
>
> `a[.][.]...[.][.][.]` (n indexes) is '`type`' (0 star); -- *sum* = *n*,
>
> `a[.][.]...[.][.]` (n-1 indexes) is '`type *`'(1 star); -- *sum* = *n*,
>
> `a[.][.]...[.]` (n-2 indexes) is '`type **`'(2 stars); -- *sum* = *n* , ... ,
>
> `a` (0 index) is '`type **..*`'(n stars). -- *sum* = *n*

- With that in mind, it is natural to use the following codes to allocate and deallocate multidimensional arrays.

```
// one-dimencional - an appetizer
// allocate
int *a = new int[size];
// deallocate
delete[]a;
```

```
// two-dimensional - row, column
// allocate
int **b = new int*[row];  // each element of b is an 'int *'
for(int i = 0; i < row; i++)
{
  b[i] = new int[column]; // each element of b[i] is an 'int'
}
// deallocate
for(int i = 0; i < row; i++)
{
  delete[]b[i];           // b[i] is a pointer point to array
                          // {b[i][0] ... b[i][column-1]}
}
delete[]b;                // b is a pointer point to an array
                          // {b[0] ... b[row]}
```

```
// three-dimensional - row, column, height
// allocate
int ***c = new int **[row];
for(int i = 0; i < row; i++)
{
  c[i] = new int **[column];  // c[i] is 'int **', 1-fold loop
  for(int j = 0; j < column; j++)
  {
    c[i][j] = new int[height];// c[i][j] is 'int *', 2-fold loop
  }
}
// deallocate
for(int i = 0; i < row; i++)
{
  for(int j = 0; j < column; j++)
  {
    delete[]c[i][j];         // c[i][j] is 'int *', 2-fold loop
  }
  delete[]c[i];              // c[i] is 'int **', 1-fold loop
}
delete[]c;
```

4.10 Using `new` / `delete` to manage dynamic structures

- The logic behind dynamic structures is muck like the logic behind dynamic simple variables or dynamic arrays. In fact, this line of thought applies to other dynamic types, class objects included.
- Basic usage:

```
struct stock{
  std::string stockName;
  double timeTag;
  double price;
};
// allocate
stock *Stock = new stock;
stock *myStocks = new stock[17];
```

```
// use. Pay attention to operator precedence rule in C++.
Stock->stockName = "Emma Photography";
(*Stock).timeTag = 111022;                // ugly but valid
*Stock.price = 20.50;                     // invalid.
myStocks[0]->stockName = "Dennis Studio";
(*myStocks[0]).timeTag = 111022;          //ugly but valid
*myStocks[0].price = 17.88;               // invalid.
```

```
// deallocate
delete Stock;
delete[]myStocks;
```

4.11 Automatic storage, static storage, and dynamic storage

- C++ has three ways of managing memory for data, namely automatic, static, and dynamic storage, depending on the method used to allocate memory. Dynamic storage is sometimes called the *heap* or the *free store*.
- Data objects allocated in these three ways differ from each other in their *storage duration* - how long they remain in existence. Chapter 9 discusses other differences among these three storage methods in more detail.
- Automatic storage
    - Ordinary variables defined inside a function use *automatic storage* and thus are called *automatic variables*. This term suggests that these variables come to existence automatically when the code block containing them is invoked, and expire when the code block terminates. On the memory's point of view, memory blocks are allocated for them automatically when the code block starts running, and are freed automatically when the code block terminates.

> Code block: a section of code between braces `{}`. e.g. a function is a code block, and a `for` or `if` statement is also a code block (inside a function).

  - Automatic variables are local to the block of code that contains them. They are invisible outside the block, and exist during the block's lifetime.
  - Automatic variables are typically stored on the *stack* of the program's memory. This mean that when program execution enters a block of code, its variables are added consecutively to the stack in memory and are freed in reverse order (*First-in-last-out*, FILO). The stack grows and shrinks as execution proceeds.
- Static storage
  - *Static storage* is storage that exists throughout the lifetime of the entire program. There are two ways to make a variable static. One is to define it externally, outside any function. The other is to use the keyword `static` when declaring a variable, e.g. `static double price = 2.5;`.
  - Static variables are stored on *static storage area*.
- Dynamic storage
  - The glamorous `new` and `delete` operators provide a more flexible approach than automatic and static variables. These two operators manage a pool of memory, which is referred to as the *heap*, or *free store*.
  - The heap is separate from the memory used for automatic and static variables.
  - The lifetime of data on the heap is determined by `new` and `delete`, and is not tied arbitrarily to the life of a code block (automatic storage) or a program (static storage). Using `new` and `delete` gives you much more control over how a program uses memory than does using ordinary variable declarations.
  - In a stack, the automatic addition and removal mechanism results in the part of memory in use always being contiguous. Bur the interplay between `new` and `delete` can leave "holes" in the heap, making keeping track of where to allocate new memory requests a little bit tricky for the computer.
  - Generally, the efficiency of using the heap is lower than that of using the stack.
  - What happens if you *do not* call delete after creating a variable on the heap with the `new` operator? The variable or construct dynamically allocated on the free store continues to persist if `delete` is not called, even though the memory that contains the pointer has been freed due to rules of scope and object lifetime. In essence, you have no way to access the construct on the heap because the pointer to the memory that contains it is gone. You have now created a *memory leak*.

    > Even the best programmers and software companies create memory leaks. To avoid them, it is best to get into the habit of joining your `new` and `delete` operators immediately, planning for and entering the deletion of your construct as soon as you dynamically allocate it. C++'s smart pointers (Chapter 16) help automate the task.

4.12 Array alternatives 1: the `vector` template class

- The `vector` template class is similar to the `string` class in that it is a dynamic array - its size can be set during runtime, and you can append new data to the end or insert new data in the middle. Because it is dynamic, objects of `vector` class are stored in the heap, just like variables whose memory are allocated by `new`.
- Though `vector` objects are stored in the heap and thus use `use` and `delete` to manage the memory, it does not need you to take care - `new` and `delete` are called automatically by `vector` objects. This feature enables you to use them just like ordinary automatic variables.
- The definition of `vector` is included in header `<vector>` and the `vector` identifier is part of the `std` namespace, as are other template classes/objects identifiers.
- Declare `vector` objects of a certain type

```
std::vector<int> vi;          // create a zero-size vector of 'int'
std::vector<double> vd(n);    // create an vector of n 'double'
std::vector<std::string> vst;// the type can be of other template class
std::vector<Car> vcar(6);     // the type it stores can be user-defined
std::vector<float> vf(2) = {1.0, 2}; // list-initialization is allowed
```

- Some methods `vector` class provides:

```
vi.push_back(2);              // append integer 2 to the vector vi;
vi.push_back(2);              // append another integer
vi.push_back(5);              // append yet another integer
int a = vi.pop_back();        // pop up the last element and delete it
                              // from vi. Now a = 3;
int b = vi.at(0);             // returns the 0th element of vi to b
int s = vi.size();            // returns the number of elements of vi
vi.clear();                   // clear vi
bool e = vi.empty();          // If vi is empty, then return true.
```

  - `vector` methods generally has safe-checking mechanism (such as avoid access violation).

4.13 Array alternatives 2: the `array` template class

- The `vector` class has more capabilities than the built-in array type, but this comes at a cost of slightly less runtime efficiency. If all you need is a fixed-size array, you might very well use built-in array types. However, as you can imagine, that has its own cost of lessened convenience and safety, which is why C++ introduced `vector` class in the first place. C++11 responded to this situation by adding the `array` template, in an attempt to eradicate the usage of built-in array types.
- An `array` object has fixed size specified during compile time, hence it uses the stack (or else static memory allocation) instead of the heap. So it shares the efficiency of built-in array types, and convenience and safety are also guaranteed.
- The definition of `array` is included in header `<array>` and the `array` identifier is part of the `std` namespace, as are other template classes/objects identifiers.

- Declare `array` objects of a certain type and size (fixed)

```
std::array<int, 5> ai;
std::array<char, 4> ac = {'a', 'b', 'k', 'q'};
std::array<Car,6> aCar;
std::array<bool, 3> ab {true, false, true}; // list-initialization
```

- Methods `array` class provides include `at()`, `front()`, `back()`, `size()`, `empty` and so forth.
  - `vector` methods generally has safe-checking mechanism (such as avoid access violation).

4.14 Pointer arithmetic

- Adding a integer to a pointer is allowed. The result of adding one equals the original address value plus a value equal to the number of bytes in the pointed-to object.
- Subtracting an integer from a pointer is also allowed. The result of subtracting one equals the original address value minus a value equal to the number of bytes in the pointed-to object.
- Subtracting a pointer from a pointer is allowed. The result of taking the difference between two pointers is the difference between the original address values divided by a value equal to the number of bytes in the pointed-to object.

4.15 Pointers and arrays mixed together

- There are two concepts that are sometimes confusing: pointer-to-array (*array pointer*) and array-of-pointer (*pointer-array*).
  - An *array pointer* (pointer-to-array) is a pointer that points to an array (not the first element of the array, but the whole array - though the address value is the same). The pointer is of type '`type (*)[size]`'. For instance,

    ```
    int a[3];
    int (*p)[3] = &a;
    ```

    Note that assigning `&a[0]` to `pa` generates an error, because the types do not match, though the value of `&a[0]` equals to the value of `&a`, as is discussed in 4.1.
  - An *pointer array* (array-of-pointer) is an array that hold one or more pointers. The array is of type '`type *[size]`'. For instance,

    ```
    int a, b, c;
    int *p[3] = {&a, &b, &c}; // the '=' can be dropped, of course
    ```

- From the two examples given above we may note that `int (*p)[3]` and `int *p[3]` have different meaning. Because brackets `[]` has higher precedence rank than asterisk `*`, the identifier `p` is associated to `[]` first unless there is a pair of parenthesis `()` to separate `p` and `[]`.
  - How about a pointer-to-pointer-to-array (ppa)?

```
// array(a): 'int [3]'
int a[3];
// pointer-to-array(pa): 'int (*)[3]'
int (*q)[3] = &a;
// pointer-to-pointer-to-array(ppa): 'int (**)[3]'
int (**w)[3] = &q;
```

- How about an array-of-pointer-to-array (apa), a pointer-to-pointer-to-array (ppa), and a pointer-to-array-of-pointer-to-array (papa)?

```
// array(a): 'int [2]'
int s[2], d[2], f[2];
// pointer-to-array(pa): 'int (*)[2]'
int (*e)[2] = &s, (*r)[2] = &d, (*t)[2] = &f;
// (1) array-of-pointer-to-array(apa): 'int (*[3])[2]'
int (*g[3])[2] = {e, r, t};
// (2) pointer-to-pointer-to-array(ppa): 'int (**)[2]'
int(**y)[2] = &e;
// (3) pointer-to-array-of-pointer-to-array(papa): int (*(*)[3])[2]'
int (*(*u)[3])[2] = &g;
```

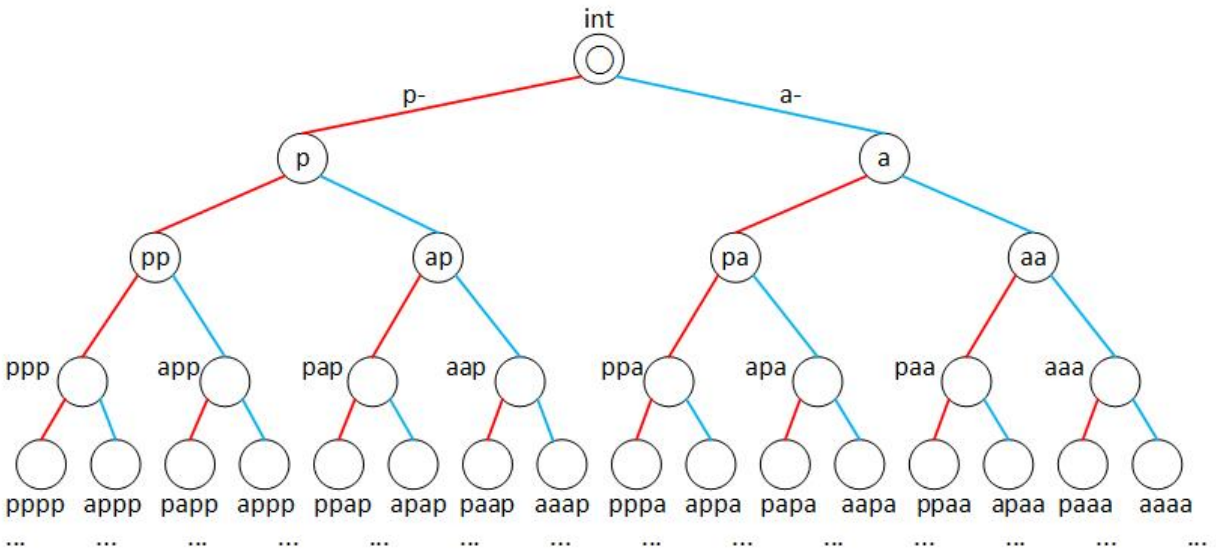
- How about more?

```
// pointer(p): 'int *'
int *i=NULL, *o=NULL, *p=NULL, *m=NULL, *n=NULL, *b=NULL;
// array-of-pointer(ap): 'int *[2]'
int *h[2] = {i, o}, *j[2] = {p, m}, *k[2] = {n, b};
// pointer-to-array-of-pointer(pap): 'int *(*)[2]'
int *(*v)[2] = &h, *(*c)[2] = &j, *(*x)[2] = &k;
// (1) array-of-pointer-to-array-of-pointer(apap): 'int *(*[3])[2]'
int *(*l[3])[2] = {v, c, x};
// (2) pointer-to-pointer-to-array-of-pointer(ppap): 'int *(**)[2]'
int *(**z)[2] = &v;
```

- How about dynamic allocation? -- *Tremble, earthlings, at Pointer's mighty power.*

```
// ppap = new pap
int *(**zz)[2] = new (int *(*)[2]);
// apap = {pap, pap, pap}
int *(*ll[3])[2] {new (int *(*)[2]), new (int *(*)[2]),
                  new (int *(*)[2])}; // list-initialization
// DO NOT forget to free the memory, of course
delete zz;
for(int i=0; i<2 ;i++) delete ll[i];
delete[]ll;
```

- How to navigate yourself through the "mess" of the pointers and arrays?



- **The golden rule** is that the precedence level of `[]` and `()` is the higher than `*`, and the associativity of these three operators is left-to-right.
- I MYSELF (July 21, 2016) have innovated an algorithm to find the correct type:
  - Suppose `g` is to be declared a variable of type $c_1c_2...c_n$, where c_i is either a 'p' or an 'a^{Xi}' (Xi is the number of array elements).
  - **STEP1:** Set arithmetic expression K = `g`, and set symbol string L = $c_1c_2...c_n$.
  - **STEP2:** Read in the first (leftmost) symbol from L and delete it from L. If this symbol is 'p', then set arithmetic expression K = p + K; if 'a^X', then set K = K × a^X. Note that the position of 'p' or 'a^X' cannot be interchanged. Since both '+' and '×' are ordinary arithmetic operators, so parentheses may be needed to keep the original order of operation. Repeat STEP2 until L is empty.
  - **STEP3:** Now we obtain the final version of arithmetic expression K. Replace 'p+' with '*', and '×a^X' with '[X]'. Leave parentheses unchanged.
  - **STEP4:** Insert the lowest-level type name, such as `int` or `MyClass`, to the left. Done.

    > This approach utilized the parallel comparison of the operator precedence relationships: '`[]`' > '`*`', '×' > '+'. The reverse algorithm can be applied to deduce the type of a compound-type variable from its declaration.

(i) Suppose you want to declare a variable of pattern a^X (array). Then

K = `g`, L = a^X → K = K×a^X = `g`×a^X, L =∅ → K = `g[X]` → K = `int g[X]`. Done. → `int g[X];`

or if you would like to use comma declaration, `int g[X], h[Y];`

(ii) Suppose you want to declare a variable of pattern *p* (pointer). Then

K = `g`, L = p $\rightarrow$ K = p+K = p+`g`, L = $\varnothing$ $\rightarrow$ K = `*g` $\rightarrow$ K = `int *g`. Done. $\rightarrow$ `int *g, *h;`

(iii) Suppose you want to declare a variable of pattern a^{X2} ppa^{X1} (array-of-pointer-to-pointer-to-array). Then

K = `g`, L = a^{X2} ppa^{X1} $\rightarrow$ K = K$\times a^{X2}$ = `g` $\times a^{X2}$, L = ppa^{X1} $\rightarrow$ K = p+K = p+`g` $\times a^{X2}$, L = pa^{X1} $\rightarrow$ K = p+K = p+p+`g` $\times a^{X2}$, L = a^{X1} $\rightarrow$ K = K$\times a^{X2}$ = (p+p+`g` $\times a^{X2}$)$\times a^{X1}$, L = $\varnothing$ $\rightarrow$ K = `(**g[X2])[X1]` $\rightarrow$ K = `int (**g[X2])[X1]`. Done. $\rightarrow$ `int (**g[X2])[X1], (**h[X2])[X1];`

(iv) Suppose you want to declare a variable of pattern pa^{X2} a^{X1} p (pointer-to-array-of-array-of-pointer). Then

K = `g`, L = $pa^{X2}a^{X1}p$ $\rightarrow$ K = p+K = p+`g`, L = $a^{X2}a^{X1}p$ $\rightarrow$ K = K$\times a^{X2}$ = (p+`g`)$\times a^{X2}$, L = $a^{X1}p$ $\rightarrow$ K = K$\times a^{X1}$ = (p+`g`)$\times a^{X2}\times a^{X1}$, L = p $\rightarrow$ K = p+K = p+(p+`g`)$\times a^{X2}\times a^{X1}$, L = $\varnothing$ $\rightarrow$ K = `*(*g)[X2][X1]` $\rightarrow$ K = `int *(*g)[X2][X1]`. Done. $\rightarrow$ `int *(*g)[X2][X1], *(*h)[Y2][Y1];`

(v) Likewise,

a. pa^{X} *(pointer-to-array)* $\rightarrow$ *(p+*`g`*)*$\times a^{X}$ $\rightarrow$ `` `int (g)[X];` ``.

b. a^{X} *p* (array-of-pointer) $\rightarrow$ p+`g` $\times a^{X}$ $\rightarrow$ `int *g[X];`.

c. a^{X2} a^{X1} (array-of-array, i.e. 2-D array) $\rightarrow$ `g` $\times a^{X2}\times a^{X1}$ $\rightarrow$ `int g[X2][X1];`.

d. a^{X2} pa^{X1} *p* $\rightarrow$ p+(p+ `g` $\times a^{X2}$)$\times a^{X1}$ $\rightarrow$ `int *(*g[X2])[X1];`.

e. pa^{X2} pa^{X1} $\rightarrow$ (p+(p+`g`)$\times a^{X2}$)$\times a^{X1}$ $\rightarrow$ `int (*(*g)[X2])[X1];`.

Utilizing C++11's automatic type deduction feature `auto`, the drudgery above can be saved...

```
// pointer(p): 'int *'
int *i=NULL, *o=NULL, *p=NULL, *m=NULL, *n=NULL, *b=NULL;
// array-of-pointer(ap): 'int *[2]'
int *h[2] = {i, o}, *j[2] = {p, m}, *k[2] = {n, b};
// pointer-to-array-of-pointer(pap): 'int *(*)[2]'
int *(*v)[2] = &h, *(*c)[2] = &j, *(*x)[2] = &k;
// pointer-to-pointer-to-array-of-pointer(ppap): 'int *(**)[2]'
int *(**z)[2] = &v;           // the "upright" way
auto z = &v;                  // the "cheating" way -- since C++11
```

... up to a point.

```
int *(*l[3])[2] = {v, c, x};  // the "upright" way
auto l[3] = {v,c,x};          // Want to cheat? Not allowed.
```

In short, keyword `auto` can be used to deduce the type of variable you initialize, using the information of the initializer. However, `auto` work for a single initializer, not for multiple initializers (at least in C++11).

The type-deduction feature provided by `auto` is compile-time deduction. It is not to be confused with *runtime type identification* (RTTI) (Chapter 15).

> A Glimpse into the Power of `auto`:
>
> The keyword `auto` is smart enough to deduce the type of a to-be-declared variable, provided the declaration has one (not multiple) initializer. That makes things easier. The code snippets below demonstrated three usages of `auto` (among other usages):
>
> ```
> // 1. deduce a variable's type (since C++11)
> auto i = 1, *pi = &i, j = i;
> ```
>
> ```
> // 2. deduce a function's return type (since C++14)
> auto function(...);
> auto function(...) {...}
>      // the declaration and the header of
>      // the definition must use auto TOGETHER.
> ```
>
> ```
> // 3. serve in the trailing return type syntax (does NOT
> //    deduce the type - just a placeholder) (since C++11)
> auto function(...) -> returnType;
> auto function(...) -> returnType {...}
>      // the declaration and the header of
>      // the definition must use auto TOGETHER.
> ```
>
> It is a philosophical shift for C++ in that in the past the compiler already possessed the ability of type deduction but it used this ability to complain when codes are wrong, whereas from now on it uses this knowledge to help you.
>
> However, `auto` might cause problems - if your code is syntactically correct but is not what you actually desire, `auto` may mask the bug. For instance,
>
> ```
> int n {1}; int *pn = &n;
> // suppose your intention is: auto q = *pn; where the variable
> // q should be of type 'int'
> auto q = &pn;  // You are wrong, but the compiler keeps silent.
> int q = &n;    // You are wrong, and the compiler complains.
> ```

> Another thing about `auto` is that it may look familiar to you. Yes, it should. In C and previous C++ standards, `auto` has an entirely different meaning - it denotes "automatic storage". But programmers rarely bother using it because `auto` can only be used to specify variables declared inside a function, yet they are automatic by default. Therefore the C++11 standard committee decided to repurpose `auto` - use it to denote type-deduction and serve in the trailing return type syntax. They chose to reuse an old keyword in that a new keyword may invalidate old codes where the word is used for other purposes.

4.16 Pointers to functions

- Functions, like data items, have addresses. A function's address is the memory address at which the machine code (0101...) for the function begins. When a function is invoked, CPU's program counter (PC) is set to the value of its address.
- Normally, it is neither important nor useful to know function's address, but it can be useful in some special occasions. For example, it is possible to write a function that takes the address of another function as an argument - enabling the first function to find the second function call and run it.

  > This approach seems more awkward than simply having the first function call the second one directly inside the first one's body, but it leaves open the possibility of passing different function addresses at different times.
  >
  > This feature may remind you of *functional*s in mathematics.

- A simple example of having functions to take other functions' address as its argument is a function used to estimate the space/time complexity of other functions.

  > Q: What is space/time complexity?
  >
  > A: Review your algorithm course notes. (Simply put, it measures the space/time efficiency of algorithms and functions - algorithms' "embodiments").

- Obtaining the address of a function: just use the function name WITHOUT trialing parentheses. In the example below, the `test()` call enables the `test()` function to invoke the `testee()` function within `test()`, whereas `check()` call first invokes the `testee()` function and then passes the return value of `test()` to the `chaeck()` function.

```
1 // suppose testee() is a function
2 test(testee);             // pass the adress of testee().
3 check(testee());          // pass the return address of testee.
```

- Declaring a pointer to a function: the core issue is, what is the type? The answer is, the type of a function is determined by the function's *signature*.

  > Function signature: the types of its arguments and, if the function is a class member, the `cons` / `volatile` qualifiers (if any) on the function itself and the class in which the member function is declared.

> (the word "signature" is a little bit confusing, especially for non-native English speakers. The word here means "trait" or "identification", not the handwritten name of a person.)

Syntax: suppose you want to declare `pf` as a pointer to function `freak()`, and the prototype of `freak()` is: `double freak(int, int);`. Then `pf` should be declare in this way: `double (*pf)(int, int);`. The pointer `pf` is of type `int (*)(int, int)`.

> Replace the function name with `*ptrName`. It is simple as that.

```
// function prototypes
double bark(int, double);
double scream(int, double);
double jump(int, int);
```

```
// pointer-to-function declaration and assignment
double (*pf)(int, double);
double (*pf)(int, double) = bark; // you can initialize it (or not).
auto pg = scream;         // use auto to declare another function pointer
double *pf(int, double); // This is NOT declaring a pointer. In stead,
                         // it is declaring a function whose return
                         // type is 'double *'. Precedence: () > *.
```

```
pf = scream;      // the pointer now points to a function
pf = jump;        // error. Mismatched function signature.
pf = scream();    // error. You can not assign 'double' to
                  // 'double (*)(int, double)'
```

- Using a pointer-function to invoke a function
  - Similar to using dereference operator `*` to obtain the value held in a block of memory a pointer points to, a function can be invoked by applying `*` to the pointer which points to the memory block where the function instructions are stored, as is shown below.

    ```
    // function declaration
    double evaluate(double *arr, int size);
    ...
    int (*pf)(double *, int) {evaluate}; // list-initialization
    double result, myArr[2] = {4, 4.5};
    result = evaluate(myArr, 2); // call the function by name
    result = (*pf)(myArr, 2);    // call the function by pointer
    result = *pf(myArr, 2);   // error. () has higher precedence than *.
    ```

  - C++ (not C) allows you to invoke the function in a neater way, in which the pointer `pf` acts as if it were a function name.

    ```
    result = (*pf)(myArr, 2);    // ordinary way

    result = pf(myArr, 2);       // neater
    ```

The ordinary way is uglier, but it provides a strong visual reminder that the code is using a function pointer.

> Q: How can `pf` and `(*pf)` be equivalent when used to invoke a function?
>
> A: One school of thought maintains that because `pf` is a pointer to a function, `*pf` is a function; hence, you should use `(*pf)` as a function call. Another school of thought maintains that because the name of a function acts like a pointer to that function (that is why the name is used to represent the function address), a pointer to that function should in turn act like the name of a function; so you should use `pf` as a function call. C++ takes the compromise view that both forms are correct, or at least can be allowed, even though they are logically inconsistent with each other.
>
> *"The ability to hold views that are not logically self-consistent is a hallmark of the human mental process"* :)

- Little trick: the signatures of these three functions are the same, and can be used interchangeably.

```
const double *f1(const double ar[], int n);
const double *f2(const double [], int);
const double *f3(const double *, int);
```

- Little trick: array of pointer-to-functions.

```
const double *(*p[3])(const double *, int) = {f1, f2, f3}; // ok
auto p[3] = {f1, f2, f3};   // NOT allowed at least in C++11.
```

Be a little playful - how about a pointer-to-array-of-pointer-to-functions? First, you may very well use `auto`. But what if we want to do it ourselves?

- Recall the algorithm set forth in 4.15. This variable `g` is essentially a pointer-to-array-of-pointer (pap). Therefore the algorithm yields `*(*g)[X]`.
- The tricky part is to add the lowest-level type. Using the example above, the final statement should be

```
const double *(*(*g)[3])(const double *, int) = &pa;
// the parenthes around *(*g)[3] is needed due to operator
// precedence law. Beacause it needs to be saparated from (...)
// after it and the '... *' ahead of it,.
```

4.17 Simplifying with `typedef`

- Apart from `auto`, `typedef` is also a useful tool to simplify declarations by setting type aliases. It also makes the program easier to understand. The following snippets demonstrate its usage.

```
// original:
double i = 1;
// becomes:
typedef double DB;
DB i = 1;
```

```
// original:
struct MP3 {std::string vendor; float price;};
MP3 myMP3;
// becomes (er... seems more complicated):
typedef struct {std::string vendor; float price;} MP3;
MP3 myMP3;
```

```
// original:
short double arr[10];
// becomes:
typedef short double DBArr[10];
DBArr arr;
```

```
int fun1(int, int *); int fun2(int, int *);
// original:
int (*pf1)(int, int *) = fun1, (*pf2)(int, int *) = fun2;
int (*pfArr[2])(int, int *) = {fun1, fun2};
int (*(*g)[2])(int, int *) = &pfArr;
// becomes:
typedef int (*pFun)(int, int *);
pFun pf1 = fun1, pf2 = fun2;
pFun pfArr[2] = {fun1, fun2};
pFun (*g)[2] = &pfArr;
```

The rule of declaring a type alias is straightforward: (1) declare a variable of that type; (2) insert the keyword `typedef` ahead of it; (3) use the variable name as the new type name.

□