Lecture Notes on C++ Multi-Paradigm Programming

黄方军

中山大学数据科学与计算机学院

2017年3月2日

Agenda

- Overview of C++
- History Notes of C++
- C++’ Extensions in procedural programming

Agenda

- Overview of C++
- History Notes of C++
- C++’ Extensions in Procedural Programming

Overview of C++

- **Except for minor details, C++ is a *superset* of the C programming language.**
- **It is a better C; supports *procedural programming*.**
- **Supports data abstraction, *object-based programming*.**
- **Supports *object-oriented programming.***
- **Supports *generic programming*（泛型编程）.**

Overview of C++

- Object-Oriented Programming
  - Encapsulation, Inheritance, and Polymorphism
  - Classes as an Abstract Data Type
  - Easy to debug and maintain
  - Mainstream in software development
  - Software components

Overview of C++

- **The most important thing to do when learning C++ is to *focus on concepts* and not get lost in *language-technical details*.**

Overview of C++

- **Your purpose in learning C++ should be:**
  - **Not simply to learn a new syntax for doing things the way you used to.**
  - **But to learn *new and better ways* of building systems.**
  - **This has to be done gradually because acquiring any significant new skill takes time and requires practice**

Overview of C++

- As you will know in later chapters, *encapsulation*, *inheritance* and *polymorphism* are the most elementary concepts to object-oriented programming.
- But knowing them doesn’t necessarily make you a good object-oriented designer.

Well, what else needed to be a good object-oriented designer or programmer?

Overview of C++

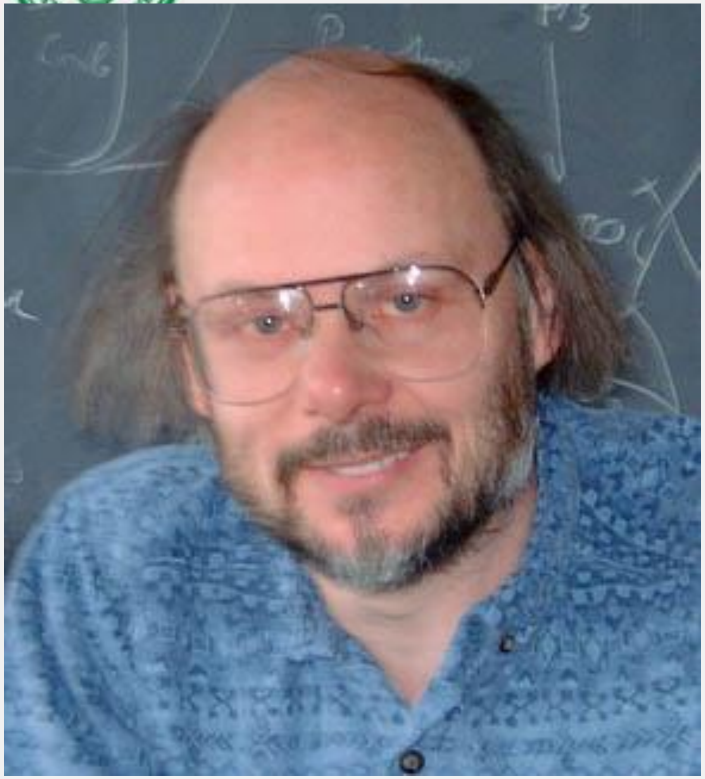
C++ is a language that you can grow with

Agenda

- **Overview of C++**
- **History Notes of C++**
- **C++' Extensions in Procedural Programming**

History Notes of C++

- **Merges notions from Sumula 67 and notions from C**
- ***1979,1980,C with Classes,* Bjarne Stroustrup at Bell Labs**
- ***1983,first C++* complier implemented**
  - **Keeps C's efficiency, flexibility and philosophy, while enjoying object-oriented programming**
- ***1985,* Cfront Release 1.0, The C++ programming language V1.0**
- ***1990,* Cfront Release 3.0, The C++ programming language V2.0**
- ***1994*, first draft of ANSI/ISO proposed standard**
- ***1997*, final draft passed, The C++ programming language V3.0**
- ***1998*, ANSI/ISO standard, ISO/IEC:98-14882**



- Bjarne Stroustrup, born in Aarhus Denmark 1950， designer and original implementer of C++ .
- Cand.Scient. (Mathematics and Computer Science), 1975, University of Aarhus Denmark. Ph.D. (Computer Science) 1979, Cambridge University, England.
- Author of The C++ Programming Language and The Design and Evolution of C++.
- College of Engineering Chair Professor in Computer Science at Texas A&M University ; member of the Information and Systems Software Research Lab, AT&T Labs – Research.
- ACM fellow. IEEE Fellow. AT&T Fellow. Member of The National Academy of Engineering.
- 1993 ACM Grace Murray Hopper award; IEEE Computer Society's 2004 Computer Entrepreneur Award; 2005 William Procter Prize for Scientific Achievement

Agenda

- Overview of C++
- History Notes of C++
- C++' Extensions in Procedural Programming

Agenda

- Overview of C++
- History Notes of C++
- C++’ Extensions in Procedural Programming
  - Line Comment （行注释）
  - Namespaces
  - C++ I/O Basics
  - Some C++ Features on Types and Variables
  - Extensions on C++ Functions
  - The new And delete Operator
  - Exception Handling

What does the C++ “hello world” Look like

```
#include <iostream>
using namespace std;
int main()
{
    cout<<“hello world!\n”;
    return 0;
}
```

```
#include <iostream.h>
int main()
{
    cout<<“hello world!\n”;
    return 0;
}
```

1-1.cpp

Line Comment

```
//this is the hello world program of C++ style
#include <iostream>
using namespace std;
int main()
{
    cout<<“hello world!\n”;
    return 0;
}
```

Agenda

- Overview of C++
- History Notes of C++
- C++' Extensions in Procedural Programming
  - Line Comment
  - Namespaces
  - C++ I/O Basics
  - Some C++ Features on Types and Variables
  - Extensions on C++ Functions
  - The new And delete Operator
  - Exception Handling

Namespaces

- *Namespaces* are used to prevent name conflicts.
- Namespace *std* is used routinely to cover the standard C++ definitions, declarations, and so on for standard C++ library.

```
namespace mfc {  //vendor 1's namespace
     int inflag;   //vendor 1's inflag
}
namespace owl {  //vendor 2's namespace
     int inflag;  //vendor 2's inflag
}
mfc::inflag = 3;  //mfc's inflag
owl::inflag = -823; //owl's inflag
```

```
using mfc::inflag;
inflag = 3;
owl::inflag = -823;
```

```
namespace mfc {  //vendor 1's namespace
      int inflag;   //vendor 1's inflag
      void g(int);
}
using mfc::inflag;   //using declaration for inflag
inflag = 100;        //OK
g(10);               //Error!
mfc::g(10);          //OK, full name
using mfc::g;        //using declaration for g
g(10);               //OK
```

```
namespace mfc {  //vendor 1's namespace
      int inflag;   //vendor 1's inflag
      void g(int);
}
using namespace mfc;   //using directive
inflag = 21;           //mfc::inflag
g(-66);          //mfc::g
owl::inflag=341;   //full name needed
```

Scope Resolution Operator

- **A hidden global name can be referred to using the scope resolution operator ::**

```
int x;
void f2( )
{
      int x = 1;    // hide global x
      ::x = 2;        // assign to global x
      x=2;     //assign to local x
}
```

but, there is no way to use a hidden local name

Agenda

- Overview of C++
- History Notes of C++
- C++’ Extensions in Procedural Programming
  - Line Comment
  - Namespaces
  - C++ I/O Basics
  - Some C++ Features on Types and Variables
  - Extensions on C++ Functions
  - The new And delete Operator
  - Exception Handling

Introduction To C++ I/O

- Still I/O is not directly a part of the C++ language. It is added as a set of types and routines found in a standard library.
- The C++ standard I/O header file is *iostream* or *iostream.h*.
- The iostream library overloads the two bit-shift operators <<, >>
- It also declares three standard streams: *cout, cin, cerr*

Introduction To C++ I/O

```
#include <iostream>
using namespace std;

int main()
{
        int x;
        int y;
        cout <<"Enter two numbers: \n";
        cin >> x >> y;
        cout <<"Their average is: ";
        cout << (x + y)/2.0 << endl;
        return 0;
}
```

Io.cpp

```
//Corresponding C style program
#include <stdio.h>
int main()
{
	int x;
	int y;
	printf("Enter two numbers: \n“);
	scanf(“%d%d”,&x,&y);
	printf("Their average is: “);
	printf(“%f\n”, (x + y)/2.0 );
	return 0;
}
```

Introduction To C++ I/O

```
#include <iostream>
using namespace std;
int main()
{
	int val;
	int sum = 0;
	cout <<"Enter next number: \n";
	while (cin >> val)
	{
		sum += val;
		cout <<"Enter next number: \n";
	}
	cout << "Sum of all values: " << sum <<
'\n';
	return 0;
}
```

Io.cpp

```
//Corresponding C style program
#include <stdio.h>
int main()
{
	int val;
	int sum = 0;
	printf("Enter next number: \n“);
	while (scanf(“%d”, &val) != EOF)
	{
		sum += val;
		printf("Enter next number: \n“);
	}
	printf("Sum of all values:%d\n “, sum );
	return 0;
}
```

Manipulators

- **Input and output can be formatted using manipulators.**
- **To use manipulators without arguments, (e.g., *endl*, *flush*, *dec*, *hex*, *left*, *right*, etc.) <*iostream*> must be included.**
- **Manipulators with arguments (e.g., *setw(n)*, *setprecision(n)*, etc.) require the header <*iomanip*>**

Manipulators

```
#include <iostream>
using namespace std;

int main()
{
      int i = 91;
      cout << "i = " << i << "(decimal)\n";
      cout << "i = " << oct << i << "(octal)\n";
      cout << "i = " << hex << i <<"(hexadecimal)\n";
      cout << "i = " << dec << i << "(decimal)\n";
      return 0;
}
```

Io.cpp

```
//Corresponding C Style Program
#include <stdio.h>
int main()
{
      int i = 91;
      printf("i = %d\n“,  i);
      printf( "i = %o  (octal)\n“, i);
      printf( "i = %h  (hexadecimal )\n“, i);
      printf( "i = %i  (decimal)\n“, i);
      return 0;
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
     int i;
     for( i = 1; i < 1000; i *= 10 )
          cout << setw(6) << i << endl;
     for( i = 1; i < 1000; i *= 10 )
          cout << i << endl;
     int a = 5;
     cout << left << setw(10) << "Karen"
           << right << setw(6) << a << endl;
     double b = 1234.5;
     cout << setprecision(2);
     cout << setw(8) << b << endl;
     return 0;
}
```

Format.cpp

```
#include <stdio.h>
int main()
{
      int i;
      for( i = 1; i < 1000; i *= 10 )
            printf(“%6d\n”, i );
      for( i = 1; i < 1000; i *= 10 )
             printf(“%d\n”, i );
      int a = 5;
      printf(“-%10s%6d\n” , "Karen“,a);
      double b = 1234.5;
      printf(“%8.2lf\n”, b);
      return 0;
}
```

"格式描述串"是由一系列的"格式转换说明符号"组成,格式转换说明符号的描述形式如下:

% [+][-] 0 m[.n] [输出精度] <形式字母>

(1)形式字母:制定输出格式,如表

d:十进制整型数

i:十进制整型数

x:十六进制整型数

o:八进制整型数

u:无符号十进制整型数

c:单个字符;

s:字符串

e:指数形式的浮点数

f:小数形式的浮点数

g:e和f中比较短的一种

p:显示变量所在的内存地址

n:它不是向printf()传递格式化信息,而是令printf()把自己已经输出的字符总数放到相应变元指

的整形变量中

%:符号%本身;

(2):输出精度如果形式字母是d,x,o.u,则可以指定如下两类精度

l:long型输出精度

h:short型输出精度

默认时为int型精度

如:long x=123454578; printf(“%d”,x);

如果形式字母为e,f,g的时候,则指定l的 时候为double精度,不指定为float精度;

(3):m[.n]指定输出长度,如果输出的是实数,则m表示该项输出占用字符位置的总长度,n表示小数部分的字符长度,如float x=4.56;printf(“%7.4f”,x);

(4)0:指定不被使用的空位置填写0,如果不指定使用0,则不使用的位置为空白.该项仅仅对数值输出时才可以指定,对字符串输出不用指定.例如

int x=234;

printf("%05d",x);//00234

printf("%5d",x);//234**

(5)[+][-]:指定输出位置,如果指定+或者缺省时为右对齐,如果为"-"的时候为左对齐;

C Style File Processing

37-62页有兴趣自己阅读

Files and Streams

- C views each file as a sequence of bytes
  - File ends with the *end-of-file* marker
  - Or, file ends at a *specified byte*
- Stream created when a file is opened
  - Provide *communication channel(通信渠道)* between files and programs
  - Opening a file returns a pointer to a *FILE* structure
    - Example file pointers:
    - stdin - standard input (keyboard)
    - stdout - standard output (screen)
    - stderr - standard error (screen)

FILE structure

- File descriptor(文件描述符)
  - Index into operating system array called the open file table
- File Control Block (FCB)
  - Found in every array element, system uses it to administer the file

Files and Streams

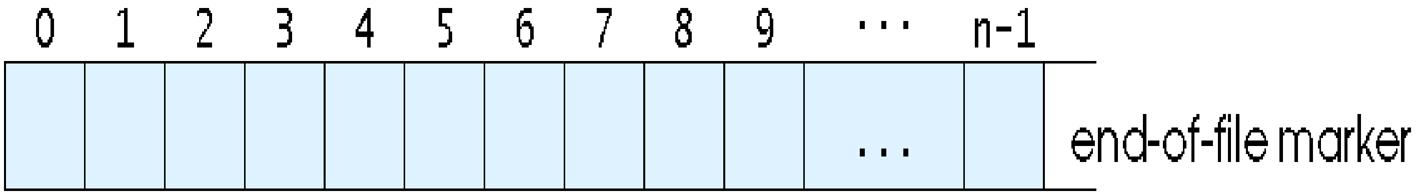


Fig. 11.2 C's view of a file of *n* bytes.

Agenda

- Data and Data Files
- Files and Streams
- Sequential Access Files
- Random Access Files

Read/Write functions(B.10)

- fgetc
  - Reads one character from a file
  - Takes a FILE pointer as an argument
  - fgetc(stdin) equivalent to getchar()
- fputc
  - Writes one character to a file
  - Takes a FILE pointer and a character to write as an argument
  - fputc('a', stdout) equivalent to putchar('a')
- fgets
  - Reads a line from a file
- fputs
  - Writes a line to a file
- fscanf / fprintf
  - File processing equivalents of scanf and printf

```
/*  Framework for Create a sequential file */
#include <stdio.h>
int main()
{
    ……
    FILE *cfPtr;   /* cfPtr = clients.dat file pointer */
    if ( ( cfPtr = fopen( "clients.dat", "w" ) ) == NULL )
        printf( "File could not be opened\n" );
    else
    {
        ……
        scanf( "%d%s%lf", &account, name, &balance );
        while ( !feof( stdin ) )
        {
                fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
                printf( "? " );
                scanf( "%d%s%lf", &account, name, &balance );
        }
        fclose( cfPtr );
    }
    return 0;
}
```

sequFile.c

Creating a Sequential Access File

- ***FILE *cfPtr;***
  - **Declares a *FILE* pointer called *cfPtr***
- ***cfPtr = fopen(“clients.dat", “w”);***
  - **Function fopen returns a *FILE* pointer to file specified**
  - **Takes two arguments – *file to open* and *file open mode***
  - **If open fails, *NULL* returned**

Creating a Sequential Access File

- *fprintf*
  - Used to print to a file
  - Like *printf*, except first argument is a *FILE pointer* (pointer to the file you want to print)
- *feof(FILE pointer)*
  - Returns true if *end-of-file* indicator (no more data to process) is set for the specified file
- *fclose(FILE pointer)*
  - Closes specified file
  - Performed automatically when program ends
  - Good practice to close files explicitly

End-of-File Key Combinations

Computer system	Key combination
UNIX systems	<return> <ctrl> d
Windows	<ctrl> z
Macintosh	<ctrl> d
VAX(VMS)	<ctrl> z

File Open Modes

Computer system	Key combination
r	Open a file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Append; open or create a file for writing at end of file.
r+	Open a file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append; open or create a file for update; writing is done at the end of the file.
rb	Open a file for reading in binary mode.
wb	Create a file for writing in binary mode. If the file already exists, discard the current contents.
ab	Append; open or create a file for writing at end of file in binary mode.
rb+	Open a file for update (reading and writing) in binary mode.
wb+	Create a file for update in binary mode. If the file already exists, discard the current contents.
ab+	Append; open or create a file for update in binary mode; writing is done at the end of the file.

Sequential Access Files

- **Also called *text file(文本文件)***
- **Each byte stores an *ASCII* code, representing a character**
- **Format of data in a *text file* is *not identical* with its format stored in memory.**

```
#include <stdio.h>
int main()
{
    ......
    FILE *cfPtr;   /* cfPtr = clients.dat file pointer */
    if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL )
        printf( "File could not be opened\n" );
    else
    {
        printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
        fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
        while ( !feof( cfPtr ) )
        {
                printf( "%-10d%-13s%7.2f\n", account, name, balance );
                fscanf( cfPtr, "%d%s%lf", &account, name, &balance );
        }
        fclose( cfPtr );
    }
  return 0;
}
```

readSequFile.c

Reading Data from a Sequential Access File

- **Create a *FILE pointer*, link it to the file to read**
  - ***cfPtr = fopen(“clients.dat", "r");***
- **Use *fscanf* to read from the file**
  - **Like *scanf*, except first argument is a *FILE pointer***
  - ***fscanf(cfPtr, "%d%s%f", &accounnt, name, &balance);***
- **Data read from beginning to end**
- ***File position pointer***
  - **Indicates number of next byte to be read / written**
  - **Not really a pointer, but an integer value (specifies byte location)**
  - **Also called *byte offset***
- ***rewind(cfPtr)***
  - **Repositions file position pointer to beginning of file (byte 0)**

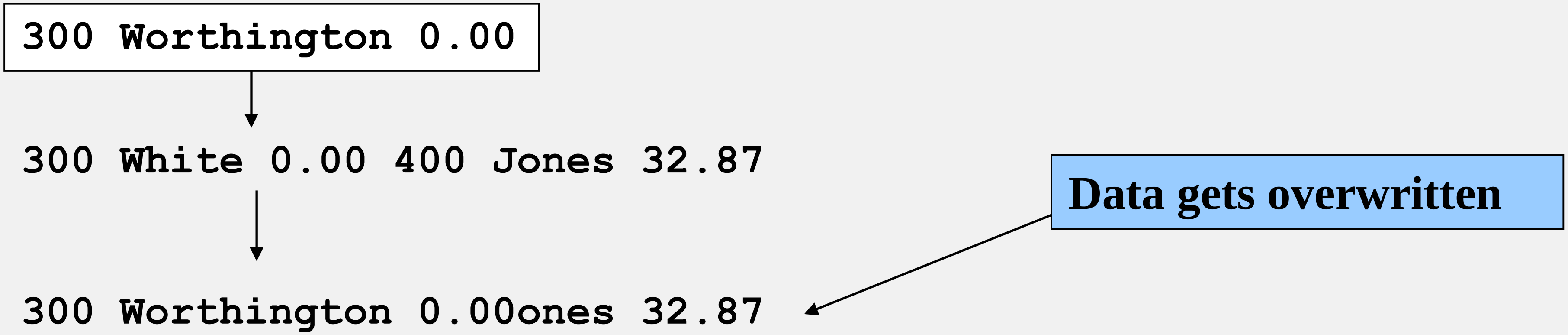
Reading Data from a Sequential Access File

- **Fields can vary in size**
  - **Different representation in files and screen than internal representation**
  - **1, 34, -890 are all *ints*, but have different sizes on disk**
- **Cannot be modified without the risk of destroying other data(修改文件内容时有可能破坏不该破坏的数据)**

Reading Data from a Sequential Access File

`300 White 0.00 400 Jones 32.87   (old data in file)`

If we want to change White's name to Worthington,



```
300 Worthington 0.00

300 White 0.00 400 Jones 32.87

300 Worthington 0.00ones 32.87
```

Data gets overwritten

Agenda

- Data and Data Files
- Files and Streams
- Sequential Access Files
- Random Access Files

Random Access Files

- **Also called *binary files(二进制文件)***
- **Format of data in a *binary file* is *identical(同样的)* with its format stored in memory.**
- ***byte doesn't necessarily represent character*; groups of bytes might represent other types of data, such as integers and floating-point numbers**
- **Records in binary files have identical length.**

Random Access Files

12345

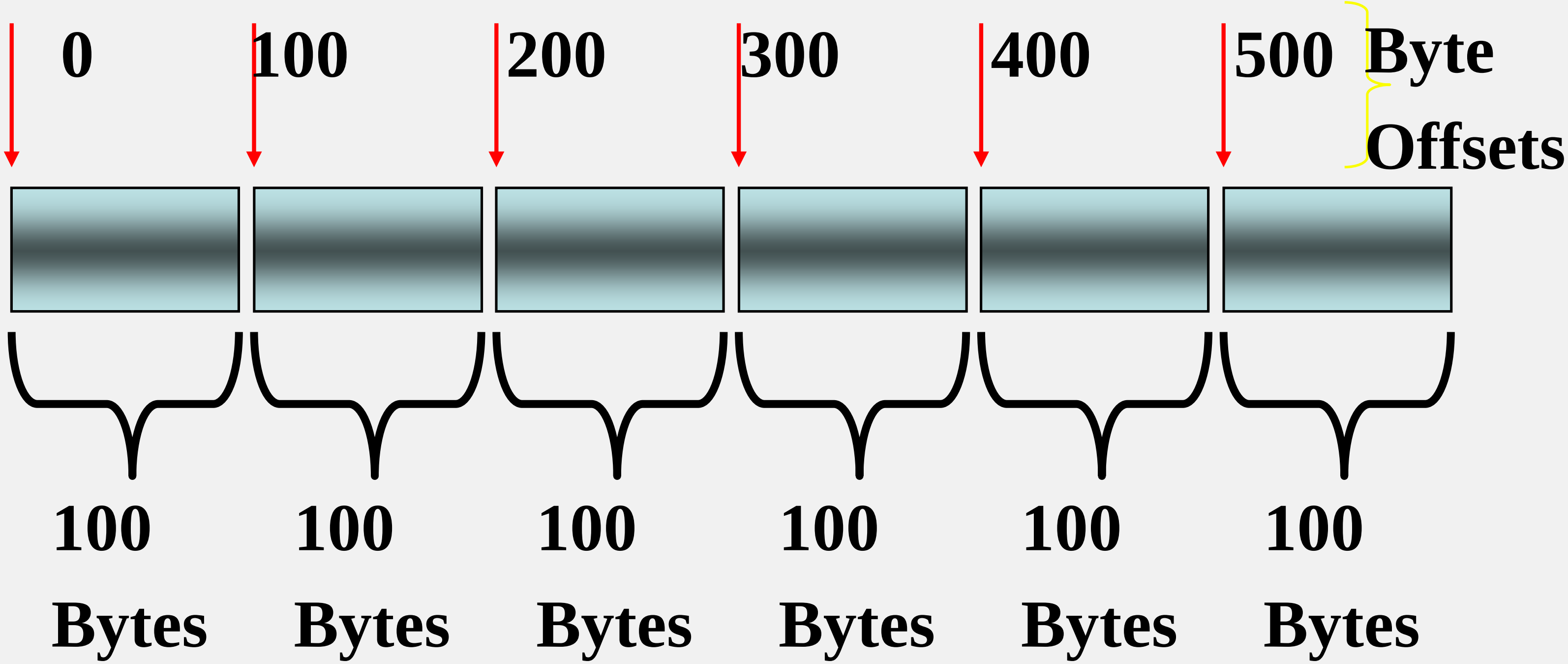
00000001	00000010	00000011	00000100	00000101

00000000	00000000	00110000	00111001

Random Access Files

- Access individual records without searching through other records
- *Instant access* to records in a file(对文件记录的随机访问)
- Data can be *inserted without destroying* other data
- Data previously stored can be updated or deleted without overwriting.

Random Access Files



```
/* Creating a randomly accessed file sequentially */
#include <stdio.h>
struct clientData {
  int acctNum;
  char lastName[ 15 ];
  char firstName[ 10 ];
  double balance;
};
int main()
{
     int i;
     struct clientData blankClient = { 0, "", "", 0.0 };
     FILE *cfPtr;
     if ( ( cfPtr = fopen( "credit.dat", "w" ) ) == NULL )
          printf( "File could not be opened.\n" );
     else
     {
          for ( i = 1; i <= 100; i++ )
                    fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
          fclose( cfPtr );
     }
     return 0;
}
```

creatRandomFile.c

Unformatted File I/O Functions

- *fwrite* - Transfer bytes from a location in memory to a file
- *fread* - Transfer bytes from a file to a location in memory

Unformatted File I/O Functions

- **fwrite(&number, sizeof(int), 1, myPtr);**
  - ***&number*** **- Location to transfer bytes from**
  - ***s izeof(int)*** **- Number of bytes to transfer**
  - ***1*** **- For arrays, number of elements to transfer**
    - ✓**In this case, "one element" of an array is being transferred**
  - ***myPtr*** **- File to transfer to or from**
  - ***fread*** **similar**

Unformatted File I/O Functions

- fwrite(&myObject, sizeof (struct myStruct), 1, myPtr);
  - To write a data block with designated size to a file
  - sizeof - Returns size in bytes of object in parentheses
- To write several array elements
  - Pointer to array as first argument
  - Number of elements to write as third argument

```
……
int main()
{
    FILE *cfPtr;
    struct clientData client = { 0, "", "", 0.0 };
    if ( ( cfPtr = fopen( "credit.dat", "r+" ) ) == NULL )
        printf( "File could not be opened.\n" );
    else
    {
        ……
        while ( client.acctNum != 0 )
        {
                printf( "Enter lastname, firstname, balance\n? " );
                fscanf( stdin, "%s%s%lf", client.lastName, \
                        client.firstName, &client.balance );
                fseek( cfPtr, ( client.acctNum - 1 ) *  \
                        sizeof( struct clientData ), SEEK_SET );
                fwrite( &client, sizeof( struct clientData ), 1,   cfPtr );
                printf( "Enter account number\n? " );
                scanf( "%d", &client.acctNum );
        }
        fclose( cfPtr );
    }
    return 0;
}
```

writeRandomFile.c

Writing Data Randomly to a Random Access File

- **int fseek(FILE *stream, long int offset, int whence);**
  - **Sets file position pointer to a *specific position***
  - ***stream* - pointer to file**
  - ***offset* - file position pointer (0 is first location)**
  - ***whence* - specifies where in file we are reading from**
    - ✓***SEEK_SET* - seek starts at beginning of file**
    - ✓***SEEK_CUR* - seek starts at current location in file**
    - ✓***SEEK_END* - seek starts at end of file**

```
……
int main()
{
    FILE *cfPtr;
    struct clientData client;
    if ( ( cfPtr = fopen( "credit.dat", "r" ) ) == NULL )
        printf( "File could not be opened.\n" );
    else
    {
        printf( "%-6s%-16s%-11s%10s\n",  \
                "Acct", "Last Name",  "First Name", "Balance" );
        while ( !feof( cfPtr ) )
        {
                fread( &client, sizeof( struct clientData ), 1, cfPtr );
                if ( client.acctNum != 0 )
                        printf( "%-6d%-16s%-11s%10.2f\n",  \
                                 client.acctNum, client.lastName, \
                                 client.firstName, client.balance );
        }
        fclose( cfPtr );
    }
    return 0;
}
```

readRandomFile.c

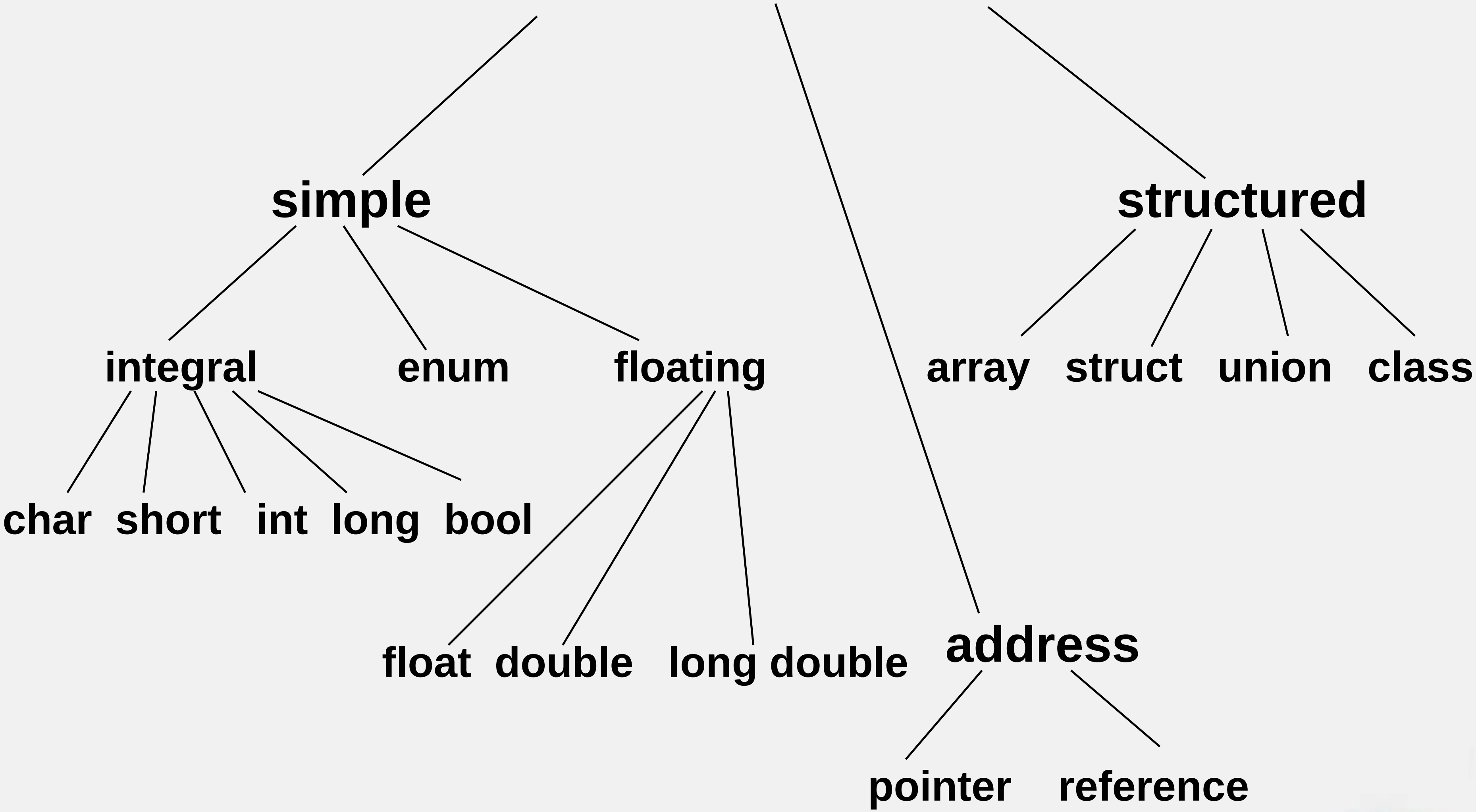
Reading Data Sequentially from a Random Access File

- fread(&client, sizeof (struct clientData), 1, myPtr);
  - Reads a specified number of bytes from a file into memory
  - Can read several fixed-size array elements
    - ✓Provide pointer to array
    - ✓Indicate number of elements to read
    - ✓Number specified in third argument

Agenda

- Overview of C++
- History Notes of C++
- C++’ Extensions in Procedural Programming
  - Line Comment
  - Namespaces
  - C++ I/O Basics
  - Some C++ Features on Types and Variables
  - Extensions on C++ Functions
  - The new And delete Operator
  - Exception Handling

C++ Data Types



simple

structured

integral enum floating

array struct union class

char short int long bool

float double long double

address

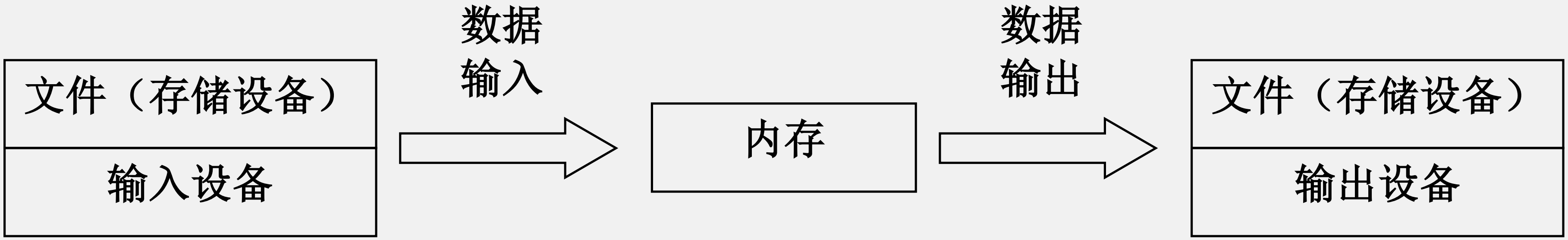
pointer reference

控制结构（control structures）

- Selection
- ➢ if-else
- ➢ switch-case
- Loop
- ➢ While
- ➢ Do-while
- ➢ For
- break, continue
- Function call

输入/输出(I/O)

- 输入（input）：从外部设备获得一个变量的值
- 输出（output）：将程序中的数据送到外部设备



文件（存储设备）

输入设备

数据输入

内存

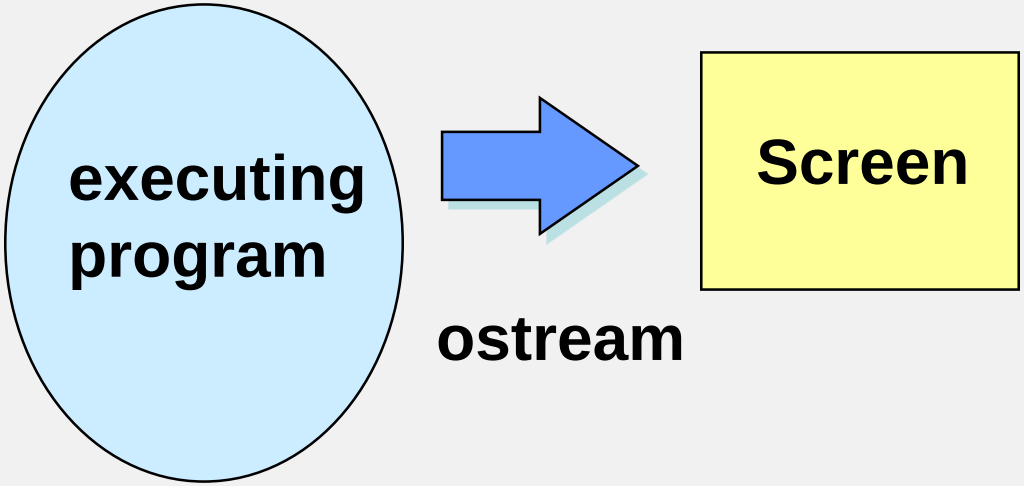
数据输出

文件（存储设备）

输出设备

No I/O is built into C++

- Instead, a library provides an output stream



cout与输出的实质

```
cout << “the answer is: ” << 3*4 << endl;
//该语句在屏幕上输出 the answer is 12
```

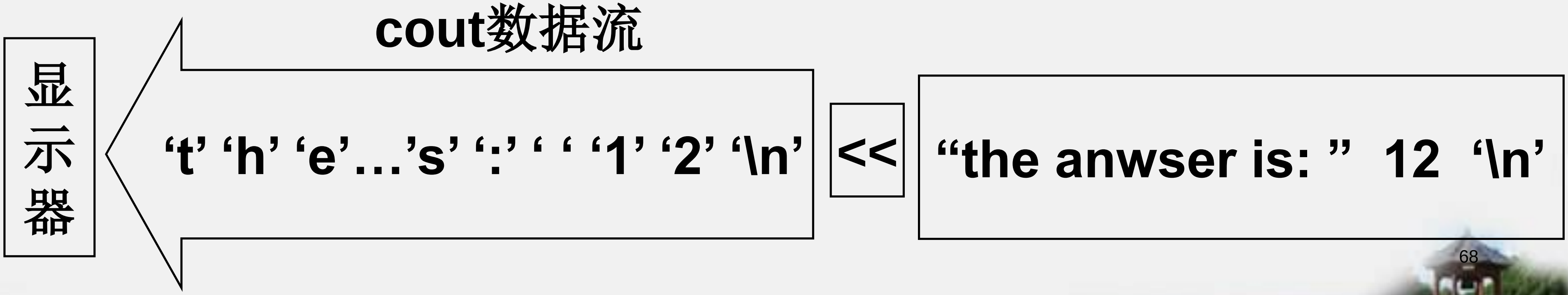
1.计算机对3*4求值得12；

2. <<把字符‘t’、’ h’ … ‘s’、’ :’、’ ‘放入cout流中；

3. <<把数值12转化为字符‘1’和‘2’，也放入cout流中

4. endl产生一个换行符，该字符也被放入cout流中

5.cout把这些字符送往显示器



显示器

cout数据流

‘t’ ‘h’ ‘e’…’s’ ‘:’ ‘ ‘ ‘1’ ‘2’ ‘\n’

<<

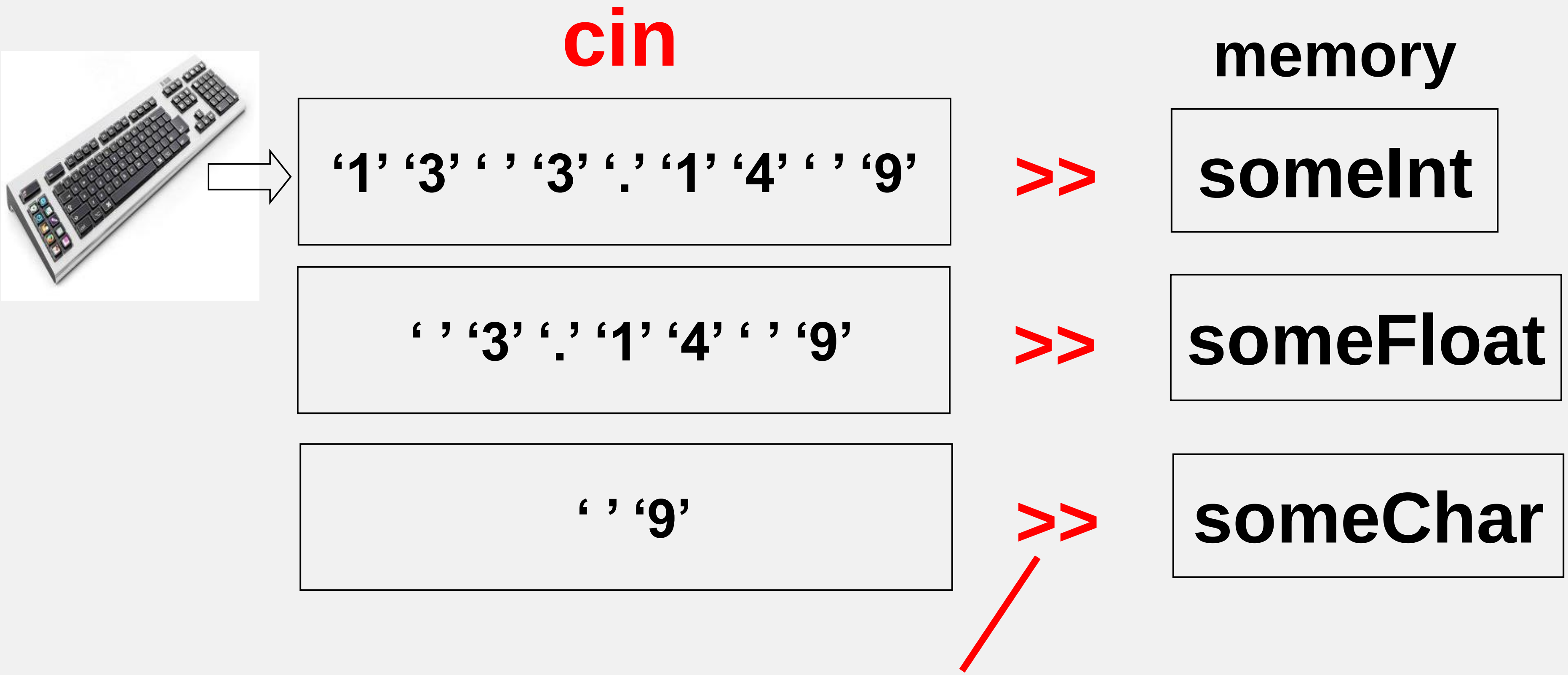
“the anwser is: ” 12 ‘\n’

cin与输入的实质

```
cin  >> someInt  >> someFloat >> someChar ;
```

1.键盘输入的字符一个一个进入输入流cin里面；
2. 一个>>代表一个输入过程。>>从cin中一个接一个获取字符，这个获取过程在哪里结束取决于变量的数据类型。该获取过程结束后，>>根据变量的数据类型，把刚才获得的字符序列转化成跟变量类型一致的数据；然后把这个数据赋给变量。
3. 下一个>>开始。

cin与输入的实质



cin

memory

‘1’ ‘3’ ‘ ’ ‘3’ ‘.’ ‘1’ ‘4’ ‘ ’ ‘9’ >> someInt

‘ ’ ‘3’ ‘.’ ‘1’ ‘4’ ‘ ’ ‘9’ >> someFloat

‘ ’ ‘9’ >> someChar

功能：读取、转化

Files

- Technique reading from and writing to (disk) files: to replace *cin* by a variable associated with an input file and to replace *cout* by a variable associated with an output file.
- Include the header *fstream* to use files.
- The operator >> is used for input in the same way that is used with *cin*, and << is used for output in the same way that it is used with *cout.*
- A variable of type *ifstream* to read from a file; A variable of type *ofstream* to write to a file

Files

```
#include <fstream>
using namespace std;
const int cutoff = 6000;
const float rate1 = 0.3;
const float rate2 = 0.6;
int main()
{
        ifstream infile;
        ofstream outfile;
        int income, tax;
        infile.open( "income.in" );
        outfile.open( "tax.out" );
        while ( infile >> income )
        {
                if ( income < cutoff )
                        tax = rate1 * income;
                else
                        tax = rate2 * income;
                outfile << "Income = " << income
                        << " greenbacks\n"
                        << "Tax = " << tax
                        << " greenbacks\n";
        }
        infile.close();
        outfile.close();
        return 0;
}
```

Income.cpp

Testing Whether Files Are Open

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
        ifstream inFile;
        ofstream outFile;
        int i;
        int j;
        inFile.open( "input.dat" );
        if( !inFile ) {
                cerr <<"Unable to open input"<< endl;
                exit(0);
        }
 ……
```

Casts

- **static_cast**
  - **Used to convert one data type to another and hands all reasonable casts**

```
average = (float) hits / (float) at_bats;
average = static_cast<float>(hits) / static_cast<float>(at_bats);
```

```
int i;
float f = 166.71;
i = static_cast<int>(f);
此时结果，i的值为166
```

Casts

- **const_cast**
  - Used to cast away constness.

```
#include <iostream>
using namespace std;
int main()
{           const int i = 100;
            const int *p = &i;
            int *q = const_cast<int*>(p);
            int j = i;
//          *p = j; //Error
            *q = j+1;
            cout << i << endl << j << endl << *p << endl << *q <<endl; //p and q point to
                                                                        //same variable
            return 0;
}
```

Casts

- **reinterpret_cast**
  - – Used to convert a pointer of one type to a pointer of another type.
  - – Implementation dependent, must be used with caution.
- **dynamic_cast**
  - – Used for casting across or within inheritance.

Constants

- **In C++, unlike in C, a const variable can be used anywhere a constant can appear.**

```
const int size = 100;
float a[size]
```

Data Type bool

- **a new so-called *built-in type* added in C++: *bool*.**

```
bool flag;
flag = ( 3 < 5 );
cout << flag << endl;
cout << boolalpha << flag << endl;
```

Enumeration

- **Once defined, an enumeration is *used like a type*, an integer type**

```
enum maritalStatus { single,
maried };
maritalStatus m;
m = single;
int sum = 0;
if( m == single ) sum++;
```

```
enum { MIN_SIZE = 0, MAX_SIZE =
100 };
int minVal = MIN_SIZE;
int arr[MAX_SIZE];
```

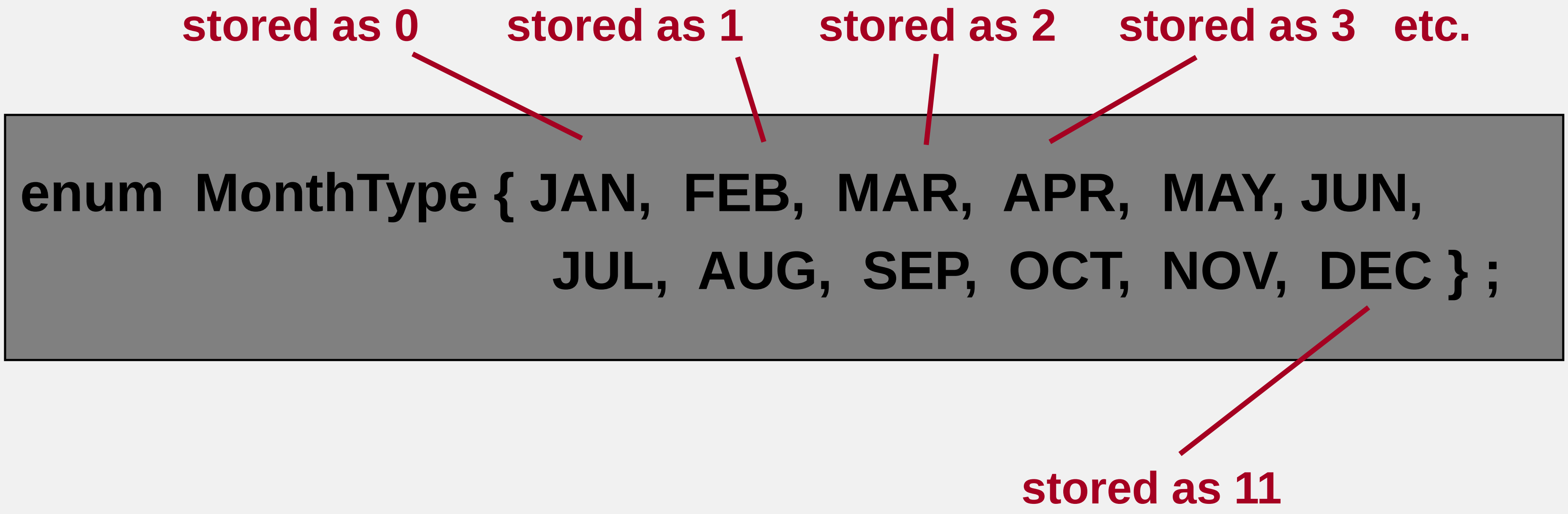
Declaring enum Type Variables

```
enum  MonthType { JAN,  FEB,  MAR,  APR,  MAY, JUN,
                  JUL,  AUG,  SEP,  OCT,  NOV,  DEC } ;

MonthType  thisMonth;                // declares 2 variables
MonthType  lastMonth;                // of type MonthType

lastMonth  =  OCT ;                  // assigns values
thisMonth  =  NOV ;                  // to these variables
          :
lastMonth = thisMonth ;
thisMonth = DEC ;
```

Storage of enum Type Variables



stored as 0 stored as 1 stored as 2 stored as 3 etc.

```
enum  MonthType { JAN,  FEB,  MAR,  APR,  MAY, JUN,
                         JUL,  AUG,  SEP,  OCT,  NOV,  DEC } ;
```

stored as 11

Increment enum Type Variables

```
enum  MonthType { JAN,  FEB,  MAR,  APR,  MAY, JUN,
                                            JUL,  AUG,  SEP,  OCT,  NOV,  DEC } ;

MonthType  thisMonth;
MonthType  lastMonth;

lastMonth  =  OCT ;
thisMonth  =  NOV ;
lastMonth = thisMonth ;

~~thisMonth = thisMonth++~~ ;                // COMPILE ERROR !

thisMonth = MonthType( thisMonth + 1) ;   // uses type cast
```

Declaring Variables

- In a C function, variable declarations must occur at the beginning of a block.
- In C++, variable declarations may occur anywhere in a block.

Structures

- ***struct*** **need not be included as part of the variable declaration.**

```
struct Point {
      double x, y;
};
Point p;
p.x = 2.0;
p.y = 1.0;
cout <<" ("<< p.x <<", "<< p.y << " ) "<< endl;
```

Structures

- **In C++, a struct can contain functions.**

```
struct Point {
      double x, y;
      void setVal(double, double);
};
Point p;
p.x = 3.1415926;
p.y = 1.0;
p.setVal(4.11, -13.090);
```

The Type string

- An alternative to C's null-terminated arrays of char.
- Use of type string requires the header *string*

```
#include <string>
using namespace std;

string s1;
string s2 = "Bravo";
string s3 = s2;
string s4( 10, 'x' );
cout << s3 << endl;
string fileName = "input.dat";
ifstream inFile;
inFile.open( fileName.c_str() );
cout << fileName.length() << endl;
```

The Type string

- An alternative to C's null-terminated arrays of char.
- Use of type string requires the header *string*

getlinedemo.cpp

Operations on `string` Variables

```
string s1 = "Object-Oriented ";
string s2 = "Programming";
string s3 = s1.substr( 7, 9 );
string s4 = s1 + s2;
cout << s4 << endl;
s1 += s2;
cout << s1 << endl;
s1.erase( 7, 9 );
cout << s1 << endl;
s1.insert( 7, s3 );
cout << s1 << endl;
s1.replace( 7, 9, "**" );
cout << s1 << endl;
```

Searching and Comparing Strings

```
int idx = s1.find( s2 );
if( idx < s1.length() )
        cout << "Found at index: "<< idx << endl;
else
        cout << "Not found" << endl;
if( s1 > s2 )
        cout <<"\""+s1+"\""<<
        " is greater than "<<"\""+s2+"\""<<endl;
else
        cout <<"\""+s1+"\""<<
        " is not greater than "<<"\""+s2+"\""<<endl;
```

- **C String**

Characters Arrays and Strings

- *Character arrays* are arrays used to store characters.
- C has no *string variable*. In C, *string variables* are represented internally as array of characters.

String and Its Terminator

- **C compiler always stores a *null character* in the byte that immediately follows the last character of a string.**
- **The *null character* is written as *\0* in string and character constants and has *0* as its ASCII code**

Characters Arrays and Strings

- **C *string* is slightly different from C *character array* resting with their initialization and internal representation in storage.**

```
char c[]="C program";
```

null character

C		p	r	o	g	r	a	m	\o

```
char c[]={'c', ' ','p','r','o','g','r','a','m','\0'};
```

String.c

```
/*  Treating character arrays as strings */
#include <stdio.h>

int main()
{
    char string1[ 20 ];                  /* reserves 20 characters */
    char string2[] = "string literal";          /* reserves 15 characters */
    int i;                               /* counter */

    printf("Enter a string: ");
    scanf( "%s", string1 );
    printf( "string1 is: %s\nstring2 is: %s\n"
        "string1 with spaces between characters is:\n", string1, string2 );

    for ( i = 0; string1[ i ] != '\0'; i++ )
        printf( "%c ", string1[ i ] );

    printf( "\n" );
    return 0;
}
```

Array5.c

```
/*  Treating character arrays as strings */
#include <stdio.h>

int main()
{
    char string1[ 20 ];                  /* reserves 20 characters */
    char string2[] = "string literal"; /* reserves 15 characters */
    int i;                                /* counter */

    printf("Enter a string: ");
    gets(string1 );
    printf( "string1 is:");
    puts(string1);
    printf("string2 is: ");
    puts(string2);

    printf("string1 with spaces between characters is:");
    for ( i = 0; string1[ i ] != '\0'; i++ )
        printf( "%c ", string1[ i ] );
    printf( "\n" );
    return 0;
}
```

Array6.c

Characters Arrays and Strings

```
#include <stdio.h>
int main ()
{
        char str[80];

        printf ("Enter a string, test using scanf\n");
        scanf ("%s",str);
        printf ("You have entered: %s\n\n", str);

        printf ("Enter another string, test using gets\n");
        fflush (stdin);
        gets (str);
        printf ("You have entered: %s\n", str);

        return 0;
}
```

String1.c

String Library Function

Prototype	Function description
char *strcpy (char *s1, const char *s2)	Copies the string s2 into the array s1. The value of s1 is returned.
char *strncpy (char *s1, const char *s2, size_t n)	Copies at most n characters of the string s2 into the array s1. The value of s1 is returned.
char *strcat (char *s1, const char *s2)	Appends the string s2 to the array s1. The value of s1 is returned.
char *strncat (char *s1, const char *s2, size_t n)	Appends at most n characters of the string s2 to the array s1. The value of s1 is returned.
int strcmp (const char *s1, const char *s2)	Compares the string s1 to the string s2. Returns 0, <0, or >0 if s1 is equal to, less than, or greater than s2, respectively.
int strncmp (const char *s1, const char *s2, size_t n)	Compares up to n characters of the string s1 to the string s2. Returns 0, <0, or >0 if s1 is equal to, less than, or greater than s2, respectively.
char *strerror (int errornum)	Maps errornum into a full text string in a system dependent manner. A pointer to the string is returned.
size_t strlen (const char *s)	Determines the length of string s.

String Library Function--strlen

- strlen *does not measure* the length of the array; instead, it returns the *actual length* of the string stored in the array.

```
#include <string.h>
int main()
{
        int k;
        char st[]="C language";
        k=strlen(st);
        printf("The lenth of the string is %d\n",k);
        return 0;
}
```

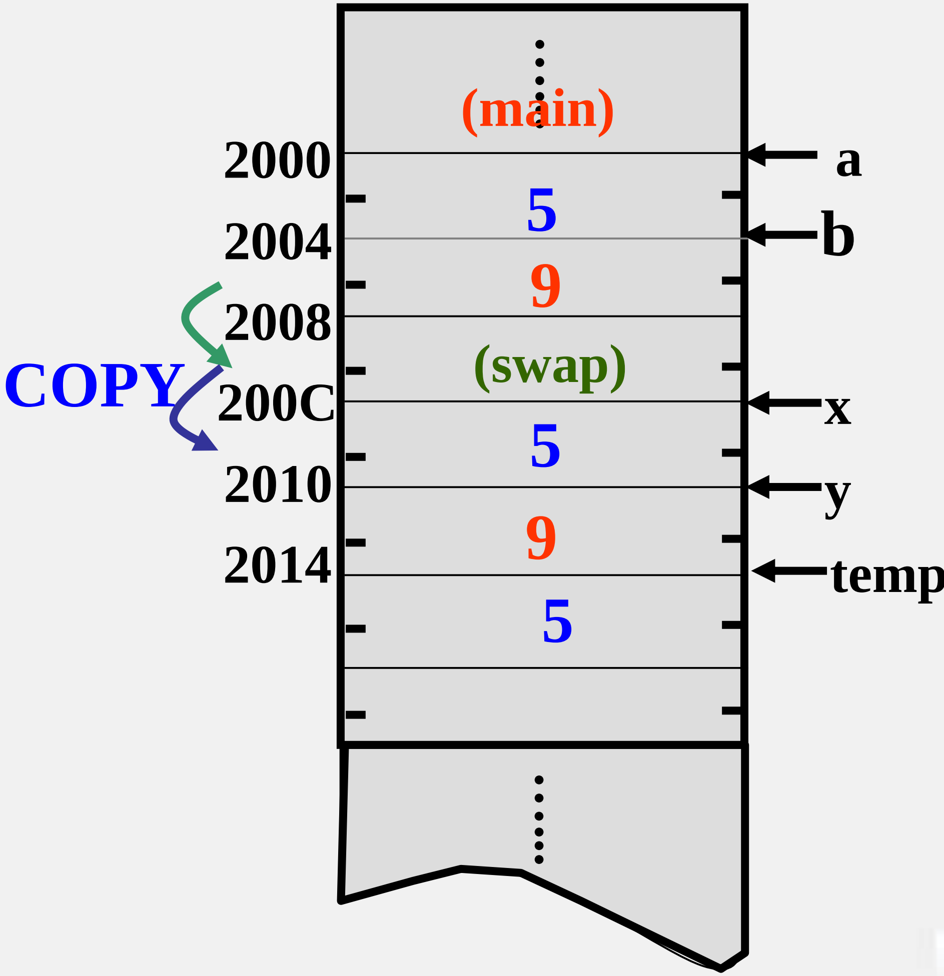
Strlen.c

Agenda

- Overview of C++
- History Notes of C++
- C++’ Extensions in Procedural Programming
  - Line Comment
  - Namespaces
  - C++ I/O Basics
  - Some C++ Features on Types and Variables
  - Extensions on C++ Functions
  - The new And delete Operator
  - Exception Handling

• Call by Value

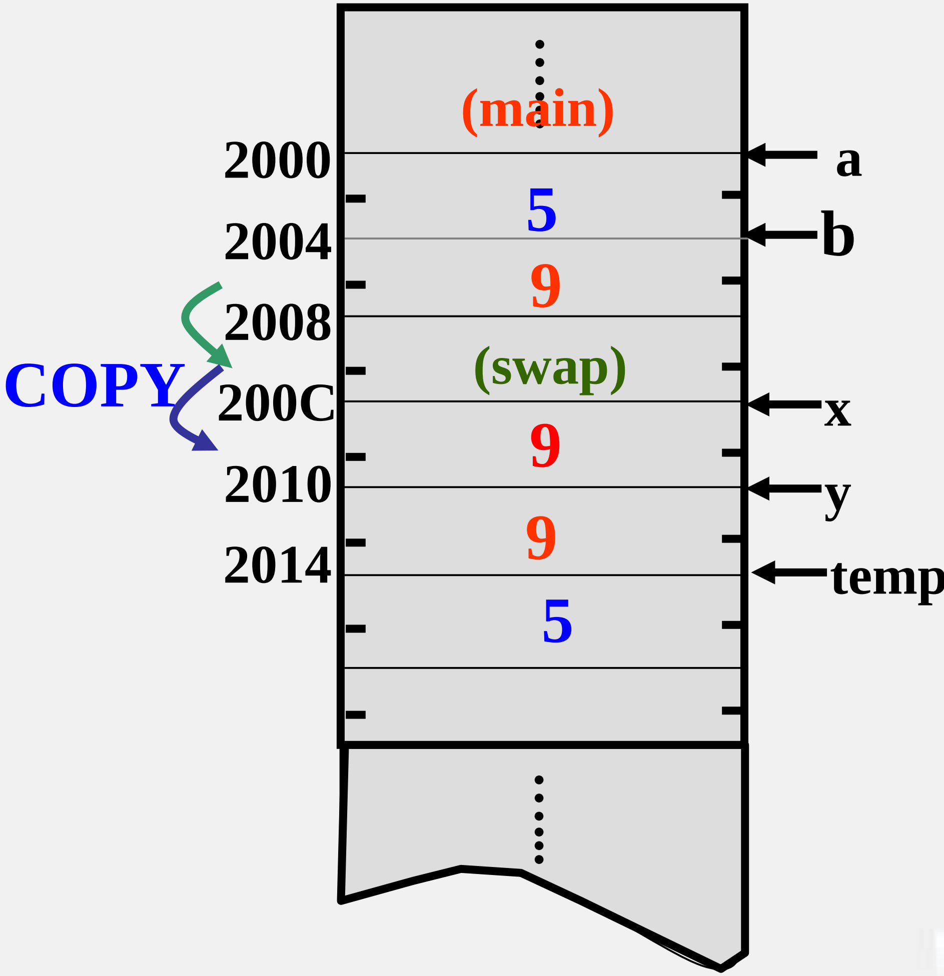
```
#include <stdio.h>
void swap(int  x,int y)
{
      int  temp;
      temp=x;
      x=y;
      y=temp;
}
int main()
{
      int a,b;
      scanf("%d,%d",&a,&b);
      swap(a,b);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```



Swap1.c

• Call by Value

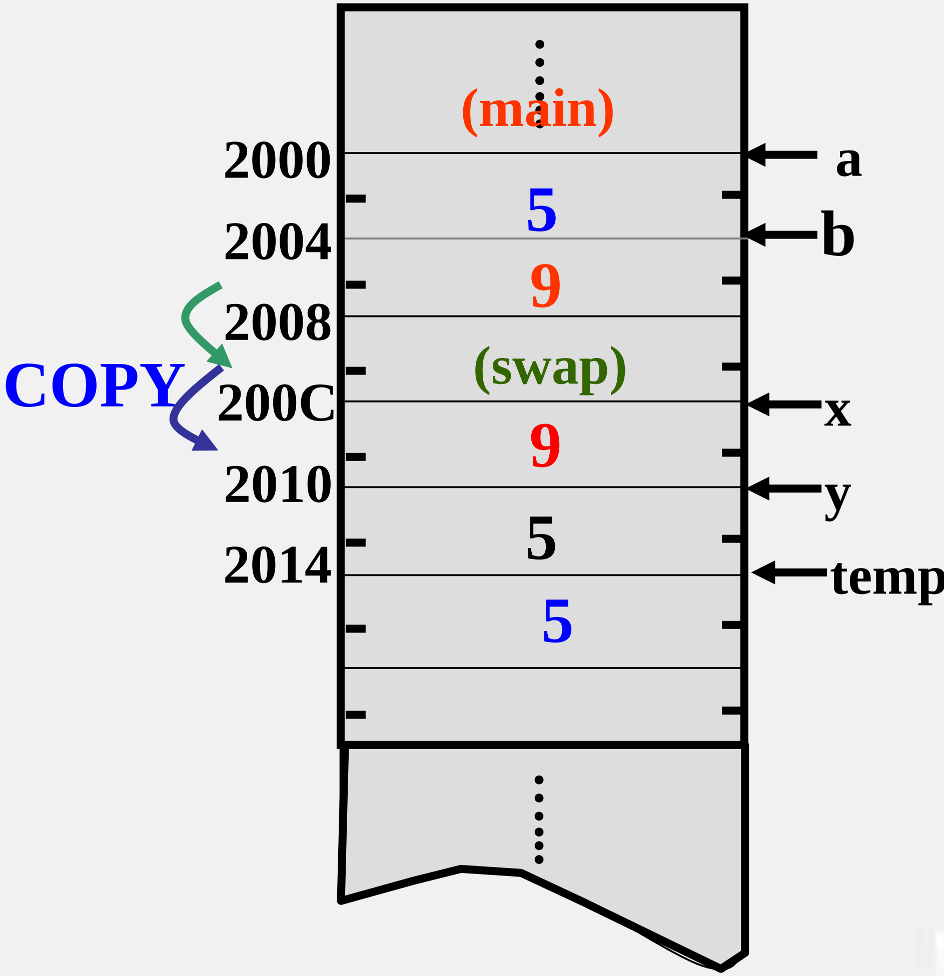
```
#include <stdio.h>
void swap(int  x,int y)
{
    int  temp;
    temp=x;
    x=y;
    y=temp;
}
int main()
{
    int a,b;
    scanf("%d,%d",&a,&b);
    swap(a,b);
    printf("\n%d,%d\n",a,b);
    return 0;
}
```



Swap1.c

- Call by Value

```
#include <stdio.h>
void swap(int  x,int y)
{
      int  temp;
      temp=x;
      x=y;
      y=temp;
}
int main()
{
      int a,b;
      scanf("%d,%d",&a,&b);
      swap(a,b);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```

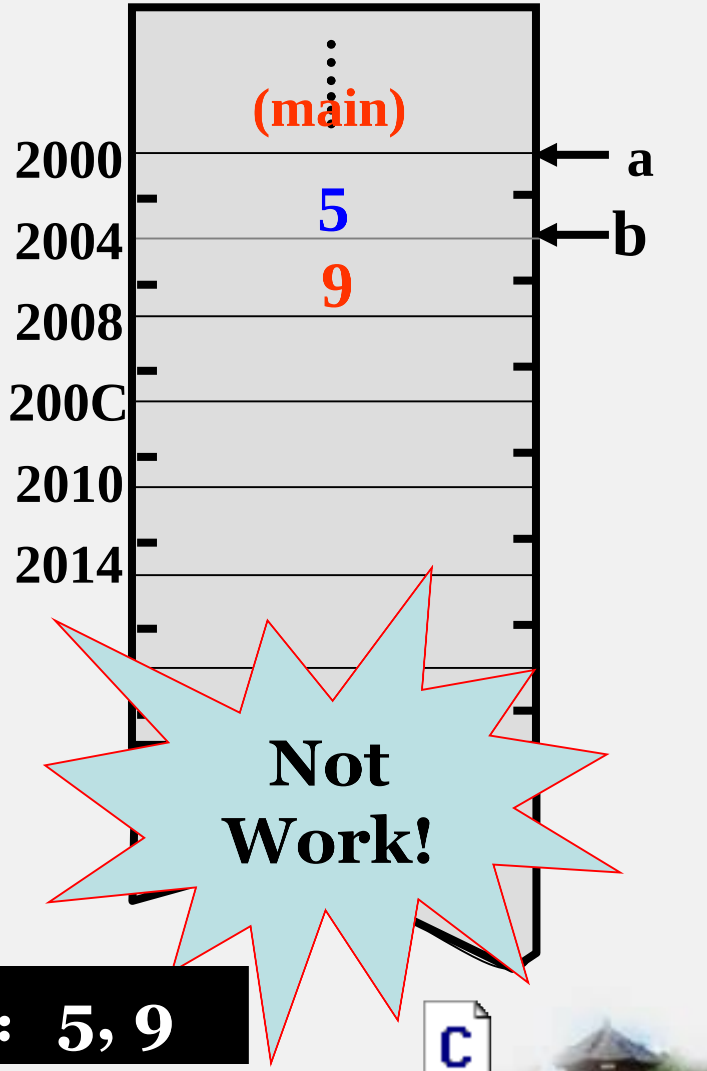


Swap1.c

- # Call by Value

```
#include <stdio.h>
void swap(int  x,int y)
{
      int  temp;
      temp=x;
      x=y;
      y=temp;
}
int main()
{
      int a,b;
      scanf("%d,%d",&a,&b);
      swap(a,b);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```

Pass by value

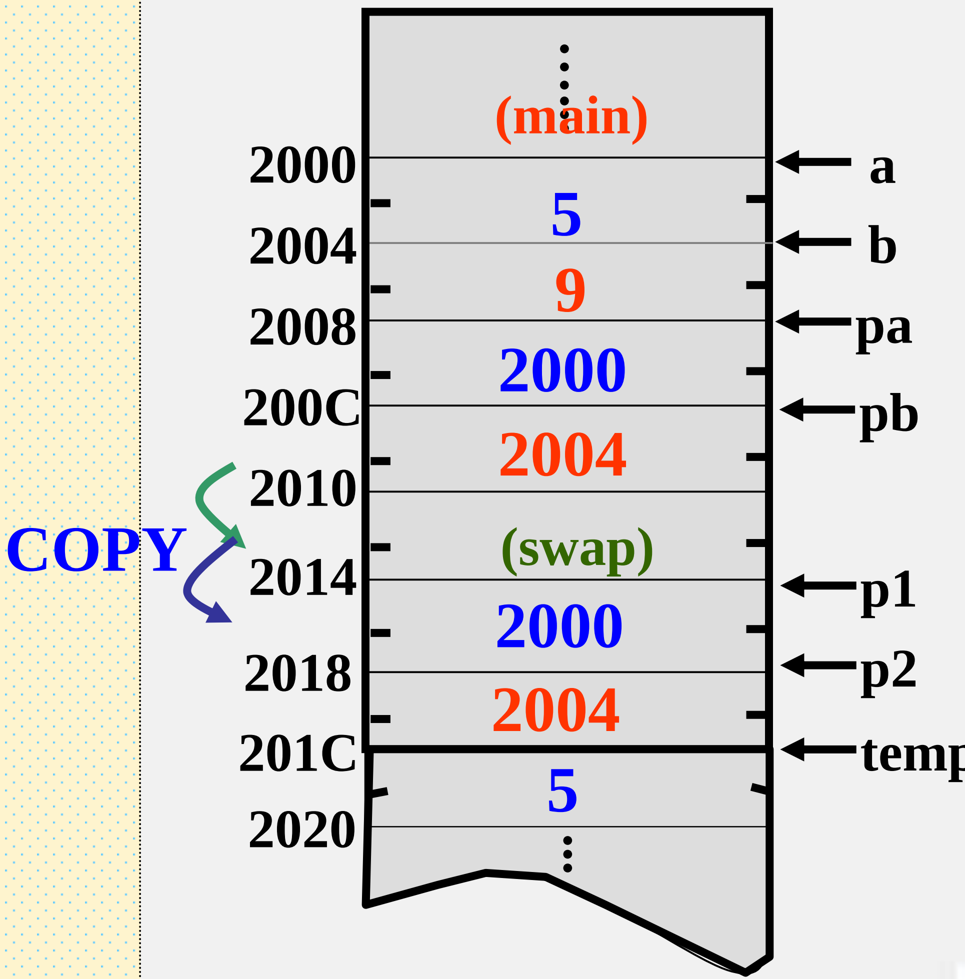


Result: 5, 9

Swap1.c

- **Call by Reference??**

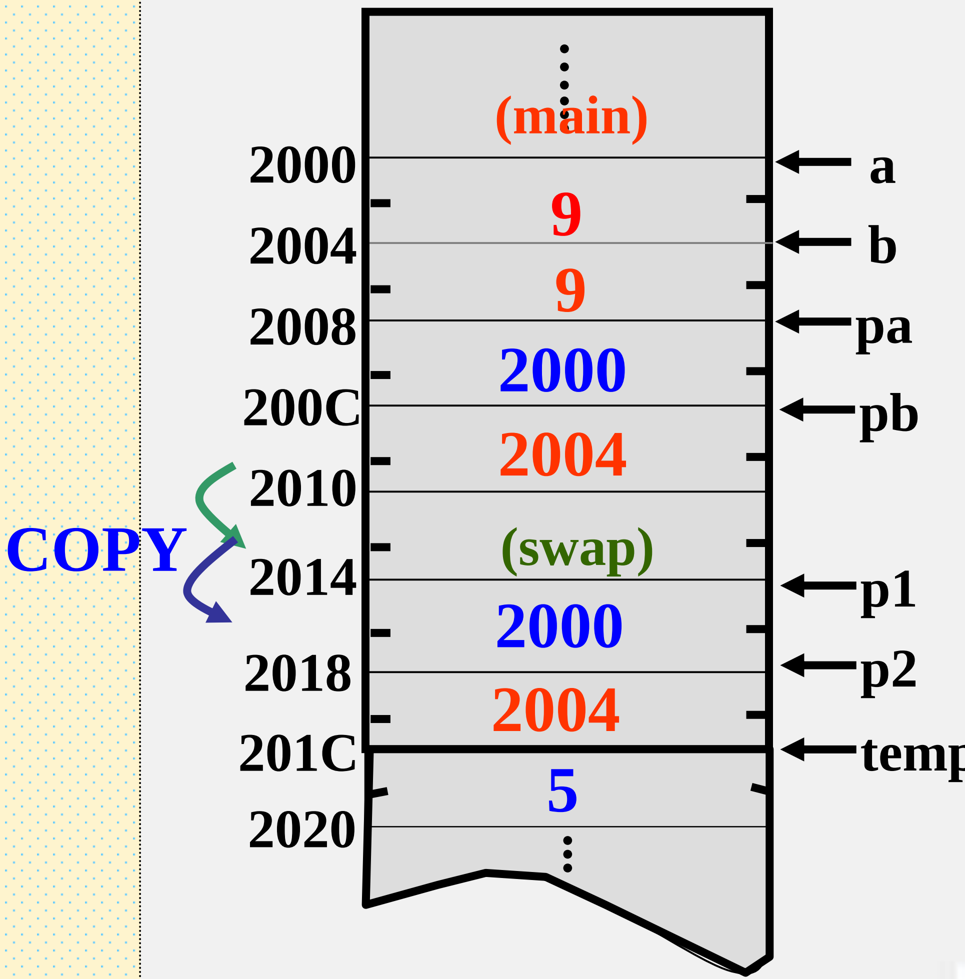
```
#include <stdio.h>
void swap(int  *p1, int  *p2)
{
      int temp;
      temp=*p1;
      *p1=*p2;
      *p2=temp;
}
int main()
{
      int a,b;
      int *pa,*pb;
      scanf("%d,%d",&a,&b);
      pa=&a;  pb=&b;
      swap(pa,pb);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```



Swap2. c

• Call by Reference??

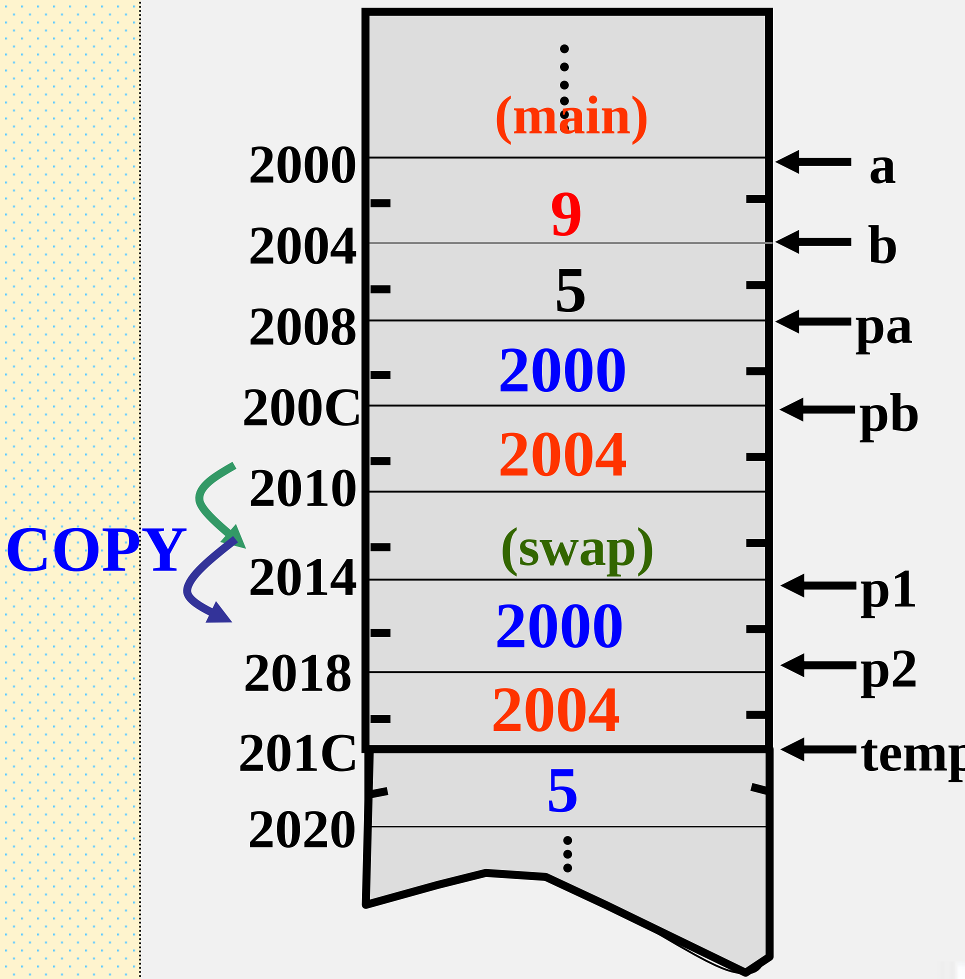
```
#include <stdio.h>
void swap(int  *p1, int  *p2)
{
      int temp;
      temp=*p1;
      *p1=*p2;
      *p2=temp;
}
int main()
{
      int a,b;
      int *pa,*pb;
      scanf("%d,%d",&a,&b);
      pa=&a;  pb=&b;
      swap(pa,pb);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```



Swap2. c

- **Call by Reference??**

```
#include <stdio.h>
void swap(int  *p1, int  *p2)
{
      int temp;
      temp=*p1;
      *p1=*p2;
      *p2=temp;
}
int main()
{
      int a,b;
      int *pa,*pb;
      scanf("%d,%d",&a,&b);
      pa=&a;  pb=&b;
      swap(pa,pb);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```



(main)
2000 9 ← a
2004 5 ← b
2008 2000 ← pa
200C 2004 ← pb
2010
COPY
(swap)
2014 2000 ← p1
2018 2004 ← p2
201C 5 ← temp
2020

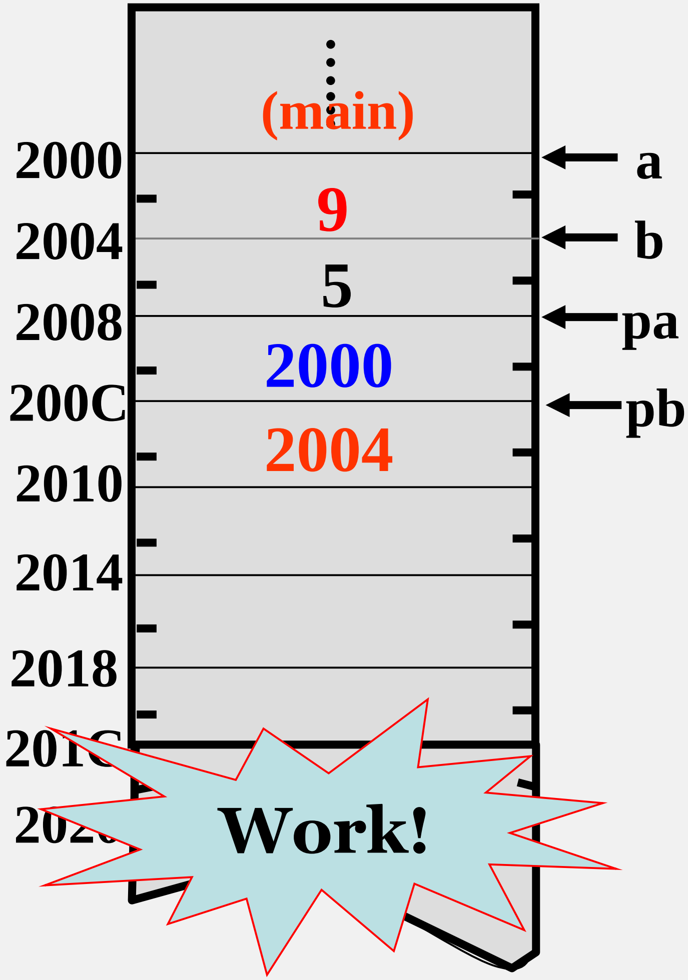
Swap2. c

• Call by Reference??

```
#include <stdio.h>
void swap(int  *p1, int  *p2)
{
      int temp;
      temp=*p1;
      *p1=*p2;
      *p2=temp;
}
int main()
{
      int a,b;
      int *pa,*pb;
      scanf("%d,%d",&a,&b);
      pa=&a;  pb=&b;
      swap(pa,pb);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```

Pass by reference

COPY



Result: 9,5

Swap2.c

例4 修改swap

```
void swap( int *p1, int *p2 )
{
     int* temp;

    *temp = *p1;
    *p1 = *p2;
    *p2 = *temp;

}
int main()
{
     swap( p1, p2 );
              …
}
```

运行出错或死机！

References

- An alternative name for an object(storage).
- For a type T, T& means reference to T.
- To ensure that a reference is a name for something, we must initialize the reference.
- The value of a reference cannot be changed after initialization; it refers to the object it was initialized to denote.

References

```
 int x = 1;
 int& ref = x;
 int& ref;            //error
// ref and x now  refer to the same int

 int y = ref;         // y = 1
 ref = 2;             // x =2
```

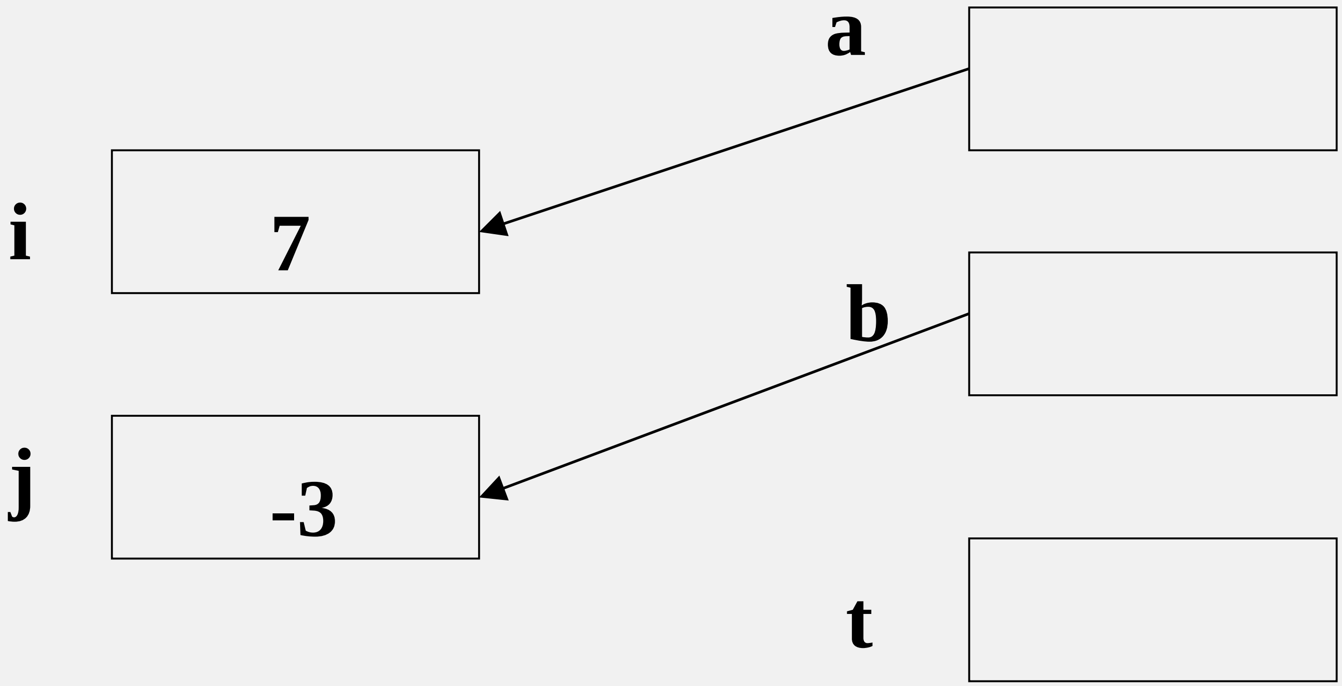
Call by Reference

```
#include <iostream>
using namespace std;
void swap(int&, int&);
int main()
{
      int i=7;
      int j=-3;
      swap(i,j);
      cout <<"i = "<< i << endl
              <<"j = "<< j << endl;
      return 0;
}
```

```
void swap(int& a, int& b)
{
      int t;
      t = a;
      a = b;
      b = t;
}
```

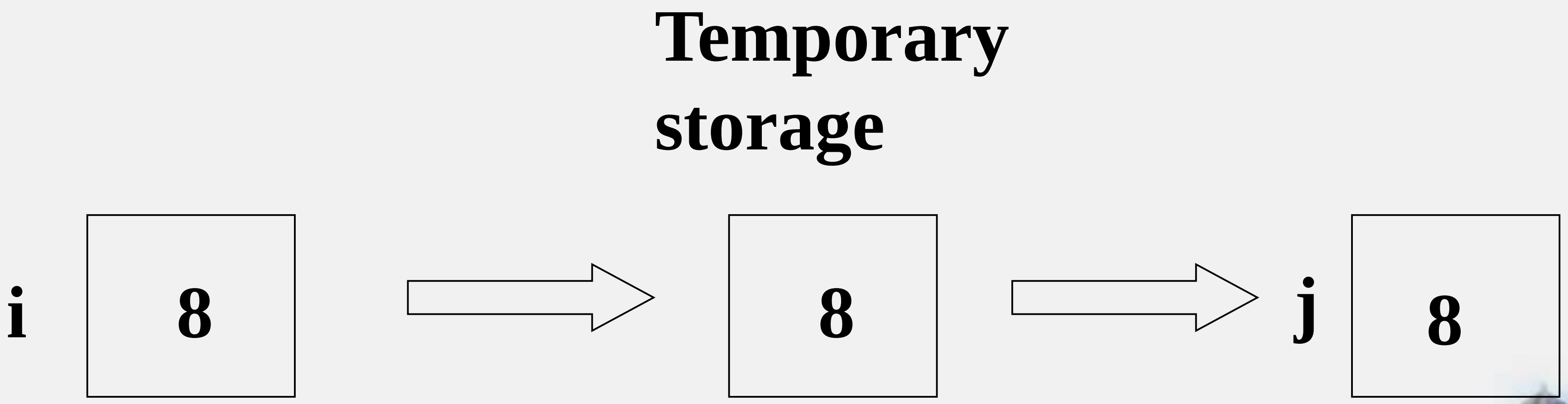
callByreference.cpp

Call by Reference



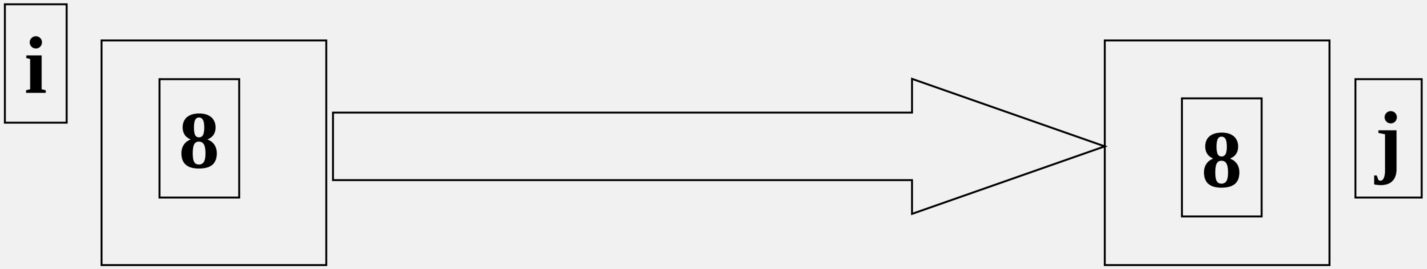
Return by Value

```
int val1()
{
      //……
      return i;
}
//……
j = val1();
```



Return by Reference

```
int& val2()
{
    //……
    return i;
}
//……
j = val2();
```



Return by Reference

```
#include <iostream>
using namespace std;
int& lastElement( int arrInt[], int size )
{
	return arrInt[size-1];
}
int main()
{
	const int ARR_SIZE = 100;
	int arr[ARR_SIZE];
	lastElement( arr, ARR_SIZE ) = 10;
	cout << arr[ARR_SIZE-1] << endl;
	return 0;
}
```

returnByReference.cpp

Returning Local Variable by References

```
int& val3()
{
      int i;
      //……
      return i; // Warning!
}
```

我们假定a的地址值为0x002345FC，那么这个0x2345FC是能够成功返回的。当return执行完成后，a就要被销毁，也就是0x002345FC所指向的内存被回收了。如果这时候在函数外面，去地址0x002345FC取值，那得到的结果肯定是不对的。这就是为什么不能返回返回局部变量的引用的道理。

Return by Reference

```
int* f(int* x)
{
        (*x)++;
        return x;
}

int& g(int& x)
{
        x++;
        return x;
}
```

Safe, x is outside this scope

```
int& h1()
{
    int q;
    return q;
}

int& h2()
{
    static int x;
    return x;
}
```

Warning! Never do this

Safe, x lives outside this scope

```
int main()
{
        int a = 0;
        f(&a); // Ugly (but explicit)
        g(a);  // Clean (but hidden)
        return 0
}
```

Same effect as in f()

Safe, outside this scope

Inline Function

```
#include <iostream>
using namespace std;
inline void swap(int& a, int& b) {
    int t;
    t = a;
    a = b;
    b = t;
}
int main() {
    int i=7, j=-3;
    swap(i,j);
    cout <<"i = "<< i << endl  <<"j = "<< j << endl;
    return 0;
}
```

Inline Function

- ***Inline* function: each occurrence of a call of the function should be replaced with the code that implements the function.**
- **However, the compiler, for various reasons, *may not be able to honor the request.***
- ***inline* functions are usually *small, frequently-used* functions.**

Preprocessor and Macro

Introduction

- The preprocessor directives fall into mainly 3 categories:
  - Macro definition. #define defines a macro; #undefine removes a macro definition.
  - File inclusion. #include directive causes the contents of a specified file to be included in a program.
  - Conditional compilation. The #if,#ifdef,#ifndef,#elif,#else and #endif directives allow blocks of text to be either included in or excluded from a program, depending on conditions that can be tested by the preprocessor
  - The remaining directives--#error,#line and #pragma are more specialized and therefore used less often.

File Inclusion

Form 1: #include <filename>

Form 2: #include "filename"

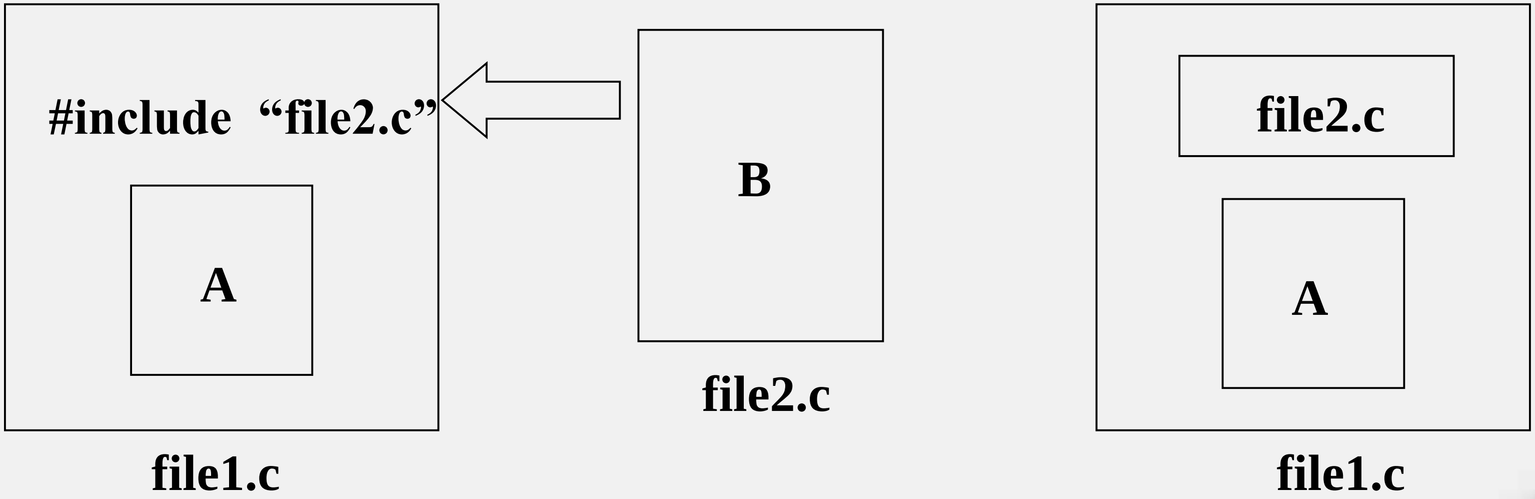
- **#include <filename> : search the system-specified directory(or directories) to find out filename.**
- **#include ”filename” : search the current directory, and then search the system-specified directory (directories).**

File Inclusion

Form 1: #include <filename>

Form 2: #include ”filename”

- #include directive tells the preprocessor to open a specified file and insert its contents into the current file.



#define: Symbolic Constants

General Form:

#define identifier *replacement-list*

- ***replacement-list*** **is any sequence of C tokens; it may include identifiers, keywords, numbers, character constants, string literals, operators, and punctuation.**
- **#define PI 3.1415926**

#define: Symbolic Constants

General Form:

#define identifier *replacement-list*

```
#define   YES      1
#define    NO      0
#define    PI      3.1415926
#define    OUT     printf("Hello,World");
```

✓

```
#define   N = 100
#define   N 100;
#define    PI        3.1415926;
```

#define: Symbolic Constants

- To distinguish from variables, macro identifier usually uses uppercases
- Its scope can refer to variables definition.
- Use *#undef* to terminate the scope of macro definition.

```
#define   YES      1
 main(){
   ........
 }
 #undef  YES
 #define   YES    0
 max(){
  ........
 }
```

Old scope
YES

New scope
YES

Symbolic Constants Advantages

- It makes programs easier to read.
- It makes programs easier to modify.
- It helps avoid inconsistencies and typographical errors.

#define: Macros

General Form:

#define identifier(x_1,x_2,…,x_n) *replacement-list*

- **x_1,x_2,…,x_n are identifiers(macro’s parameters). The parameters may appear as many times as desired in the *replacement-list.***
- **There must be no space between the macro name and the left parenthesis!**

#define: Macros

```
#include <iostream>
#define PI 3.1415926
#define CIRCLE_AREA(x) PI*x*x
using namespace std;

int main()
{
    cout << CIRCLE_AREA(1000)<<endl;
    double area， c=5;
    area=CIRCLE_AREA(4);
    area=CIRCLE_AREA(c+2);
    system(”pause”);
    return 0;
}
```

Differences Between Macros and Functions

- Macros
  - ➢Preprocessor directly inserts codes corresponding to a macro into the program
    - ✓The compiled code will often be larger
    - ✓Have no overhead caused by function call
  - ➢arguments aren’t type-checked.
  - ➢may evaluate its arguments more than once.

Conditional Compilation

Form 1:

```
#if constant_expression
        statements;
#else
        statements;
#endif
```

```
#define DEBUG 1
#include <stdio.h>
main()
{
      float c,r,s;
      printf ("input a number:  ");
      scanf("%f",&c);
#if DEBUG
      r=3.14159*c*c;
      printf("area of round is: %f\n",r);
#else
      s=c*c;
     printf("area of square is: %f\n",s);
#endif
      return 0;
}
```

Conditional Compilation

Form 2:

```
#ifdef identifier
        statements;
#else
        statements;
#endif
```

```
#include <stdio.h>
#define DEBUG
int main()
{
    int i,j;
    ……
#ifdef DEBUG
    printf(“Value of i: %d\n",i);
    printf(“Value of j: %d\n",j);
#endif
    ……
    return 0;
}
```

Conditional Compilation

Form 3:

```
#ifndef identifier
        statements;
#else
        statements;
#endif
```

The # Operator

- The # operator, which is generally called the *stringize operator*, turns the argument it precedes into a quoted string. It can appear only in the replacement list of a parameterized macro.

```
#include <stdio.h>
#define mkstr(s) #s
#define PRINT_INT(x) printf(#x”=%d”,x)
int main()
{
      int i=10,j=2;
      printf(mkstr(I like C programming));
      PRINT_INT(i/j);
      return 0;
}
```

```
printf(“I like C programming”);
```

```
printf(“i/j”“=%d”,i/j);
printf(“i/j=%d”,i/j);
```

The #,## Operator

- The ## operator, called the *pasting operator*, concatenates two tokens.

```
#include <stdio.h>
#define concat(a,b)  a##b
int main()
{
     int xy=10;
     printf("%d",concat(x,y));
     return 0;
}
```

```
printf("%d",xy);
```

The #,## Operator

- If these operators seem strange to you, keep in mind that they are not needed or used in most C programs. They exist primarily to allow the preprocessor to handle some special cases.

Inline Function V.S. Macro

- Similarities
  - Each occurrence is *replaced* with the definition.
  - The overhead of a function call is avoided so that the program may execute *more efficiently*.
  - The size of the executable image can become quite *large* if the expansions are large or there are many expansions.

Inline Function V.S. Macro

- **Dissimilarities**
  - **A macro is expanded by the *preprocessor*, an inline function is expanded by the *compiler*.**
  - **Macro expansions do text substitution *without regard to the semantics* of the code; but inline function expansions *take into account the semantics*.**
    - **Macro:No type-safety checking.**
    - **Macro:More than once parameter evaluation.**
  - **Inline functions are *generally preferable* to macros.**

Default Arguments

```
#include <string>
using namespace std;
void fo( int val,  float f = 12.6,  char c = '\n', string msg = "Error" )
{
    return;
}
int main()
{
    fo( 14, 48.3f, '\t', "OK" );
    fo( 14, 48.3f, '\t' );
    fo( 14, 48.3f );
    fo( 14 );
    return 0;
}
```

Default Arguments

```
//***** ERROR: Invalid mix of default
// and nondefault values ***

void g( int val = 0, float s, char t = ‘\n’, string msg = “error” );
```

Default arguments may only be provided for trailing arguments only

Overloading Functions

```
#include <iostream>
#include <iomanip>
using namespace std;
void print(int a);
void print(double a);
int main()
{
	int x = 8;
	double y = 8;
	print(x);
	print(y);
	return 0;
}
```

```
void print(int a)
{
	cout << a << endl;
}
void print(double a)
{
	cout << showpoint << a <<
endl;
}
```

functionOverloading.cpp

Overloading Functions

- ***Function Overloading*: using the *identical name for multiple meanings* of a function or an operator.**
- **Function overloading match resolution**
  - **Parameter type**
  - **Parameter number**
  - **Function type**

- **Match resolution**
  - **Parameter type**

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);

void h(char c, int i, short s, float f)
{
        print(c);      // exact match: invoke print(char)
        print(i);      // exact match: invoke print(int)
        print(s);      // integral promotion: invoke print(int)
        print(f);      // float to double promotion: print(double)

        print(‘a’);    // exact match: invoke print(char)
        print(49);      // exact match: invoke print(int)
        print(0);        // exact match: invoke print(int)
        print(“a”);  // exact match: invoke print(const char*)
}
```

- **Match resolution**
  - **Parameter number**

```
int pow(int, int);
double pow(double, double);
complex pow(double, complex);
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);
void k(complex z)
{
        int i = pow(2,2);             // invoke pow(int,int)
        double d = pow(2.0,2.0); // invoke pow(double,double)
        complex z2 = pow(2,z);    // invoke pow(double,complex)
        complex z3 = pow(z,2);    // invoke pow(complex,int)
        complex z4 = pow(z,z);    // invoke pow(complex,complex)
        double d = pow(2.0,2);    // error: pow(int(2.0),2)  or
                        //pow(2.0,double(2))?
}
```

- **Match resolution**
  - **Function type?**

Function Signatures

- **Overloaded functions must have distinct *signatures*.**
- **A function's *signature* consists of**
  - **Function name**
  - **The number, data types, and order of arguments**
- **Functions can not be distinguished by return types alone.**
- **Examples:**

```
void m(double, int);
void m(int, double);
double m(int, double);
```

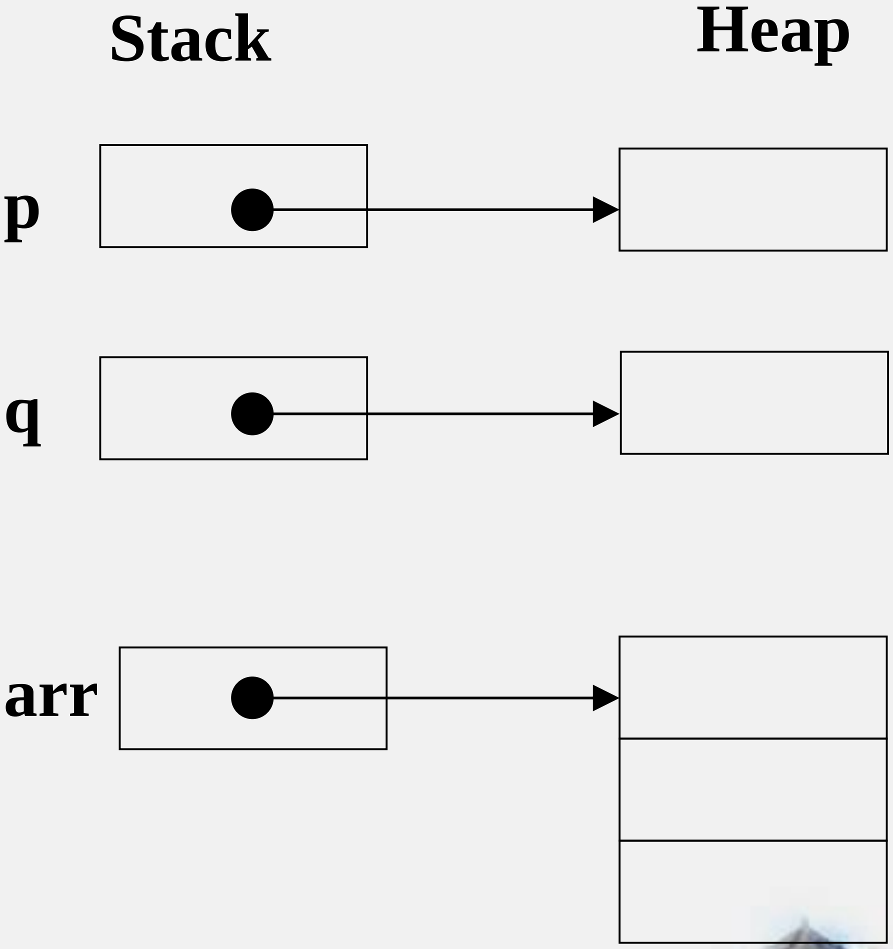
The new And delete Operators

- **new operator : creating an object on the *free store (heap)* independent of the scope**
- **delete operator : destroy the object**

```
int* p;
int* q;
p=new int(5);//allocation and initialization,*p=5
q=new int[10]; //gets q[0] to q[9] with q=&q[0]
delete p;
delete []q;
```

The new And delete Operators

```
int* p;
int* q;
p = new int;
q = new int;
*p = 40;
*q = *p;
q = p;
int* arr = new int [3];
arr[0] = 3;
arr[1] = *p;
arr[2] = 4;
delete q;
delete p; // Error!
delete [] arr;
```



The new And delete Operators

Elephant.cpp

Exception Handling

- An exception is a *run-time error* caused by some abnormal condition:
  - Out-of-bounds index
  - *new* operation fails
  - ……

异常处理部分（150-154页）后面章节专门讲解

Exception Handling

```
void g()
{
      try {
            f();  // code that may throw exception
      }
      catch ( int x ) {
            // code to handle a thrown int
      }
      catch ( char s ) {
            // code to handle a thrown char
      }
      // other catch blocks
}
```

Exception Handling

```
string s = "Object-Oriented Programming";
int index;
int len;
cout << s << endl;
while( true )
{
    cout <<"Enter index and length to erase: ";
    cin >> index >> len;
    try {
        s.erase( index, len );
    } catch ( out_of_range ) {
        cout << "Erase Error\n";
        continue;
    }
    break;
}
```

exceptionThrow.cpp

Exception Handling

```
#include <iostream>
using namespace std;
int main()
{
    int* ptr;
    try {
        ptr = new int;
    } catch ( bad_alloc ) {
        cerr <<"new: unable to allocate"<<
             " storage...aborting\n";
        exit( EXIT_FAILURE );
    }
    delete ptr;
    return 0;
}
```

Exception Handling

```
const int MAX_SIZE = 1000;
float arr[ MAX_SIZE ];
enum outOfBounds {UNDERFLOW, OVERFLOW};
float& access( int i )
{
        if( i < 0 ) throw UNDERFLOW;
        if( i > MAX_SIZE ) throw OVERFLOW;
        return arr[i];
}
```

exceptionThrow1.cpp

```
try {
        val = access( k );
} catch ( outOfBounds t ) {
        if( t == UNDERFLOW ) {
                cerr <<"arr: underflow...aborting\n";
                exit( EXIT_FAILURE );
        }
        if( t == OVERFLOW ) {
                cerr <<"arr: overflow...aborting\n";
                exit( EXIT_FAILURE );
        }
}
```

Points

- **Actually, C++ is much more simply a *superset* of C. It provides some mechanisms to serve completely different designing and programming paradigms.**
- **Above all extensions, the most critical could be abbreviated in two keywords: class and template**

Critical Points

- *Macros* are almost never necessary in C++.
- Use *const* or *enum* to define manifest constants; *inline* to avoid function-calling overhead; *templates* to specify families of functions and types; and *namespaces* to avoid name clashes.

Summary

本章重要内容：

- 引用类型
- 函数重载
- new 和 delete 操作符
- inline 函数，其与non-inline函数和Macro的区别
- 命名空间namespace

思考题

- 引用数据类型的主要作用是什么，运行时由它声明的变量会获得新的内存空间吗？
- C++函数重载的匹配规则是什么？
- C++ inline函数和普通函数以及C中的宏的关系和区别是什么？