中山大學 软件学院
SOFTWARE SCHOOL OF SUN YAT-SEN UNIVERSITY

Lecture Notes on
C++ Multi-Paradigm Programming

Bachelor of Software Engineering, Spring 2014

Wan Hai

whwanhai@163.com

13512768378

Software School, Sun Yat-sen University, GZ

ftp://stu_wh:cpp_2012@my.ss.sysu.edu.cn

From C To C++

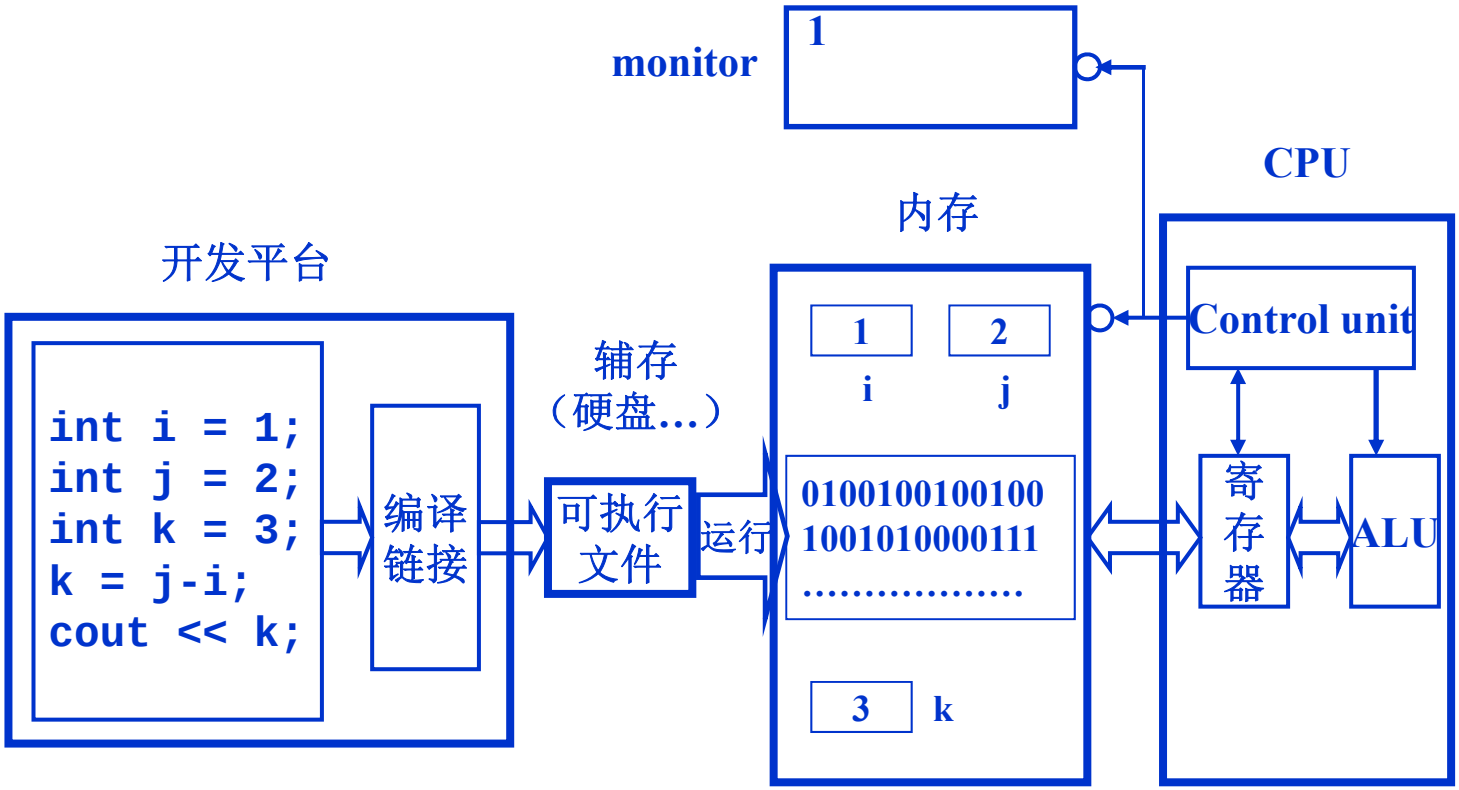
Agenda

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

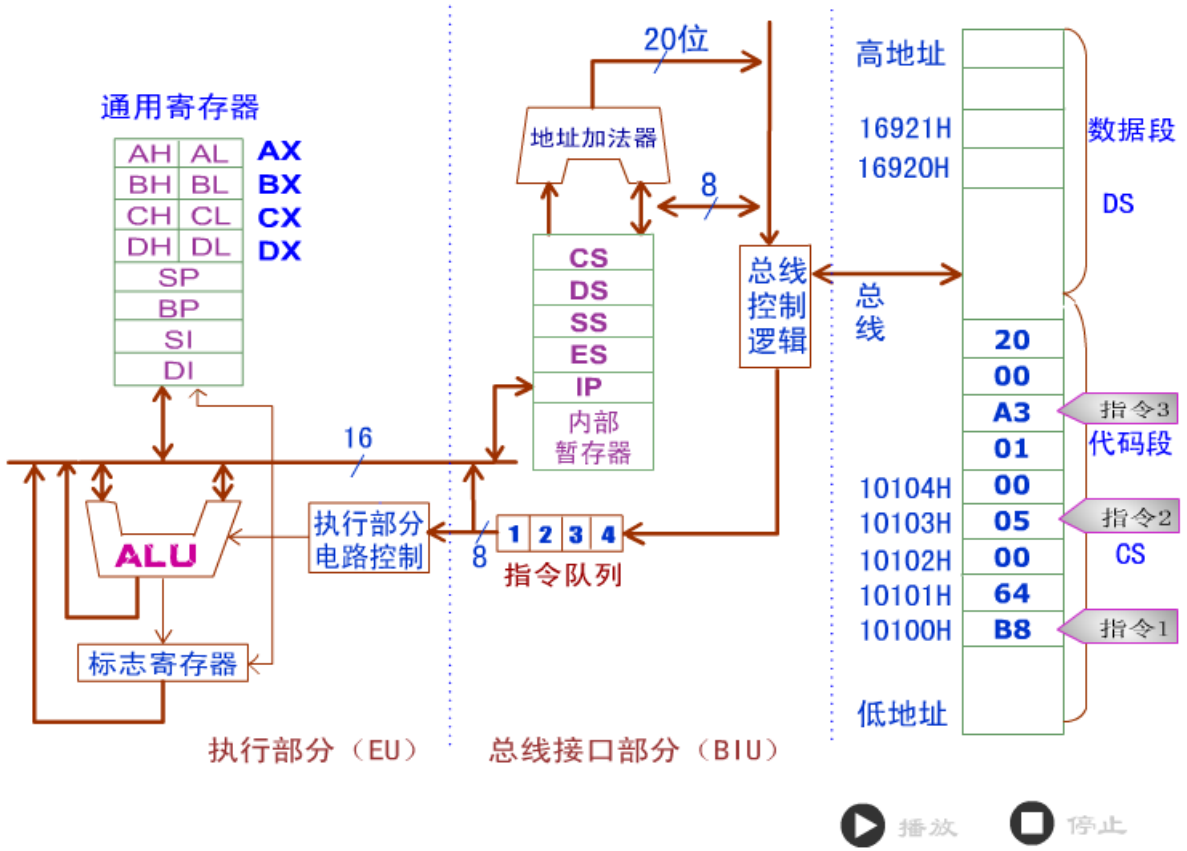
Agenda

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

Editing, compilation(linking) and execution



Why is C++


通用寄存器
AH AL AX
BH BL BX
CH CL CX
DH DL DX
SP
BP
SI
DI
20位
地址加法器
8
CS
DS
SS
ES
IP
内部暂存器
总线控制逻辑
总线
16
ALU
执行部分电路控制
标志寄存器
8
1 2 3 4
指令队列
执行部分（EU）
总线接口部分（BIU）
高地址
16921H
16920H
数据段
DS
20
00
A3
指令3
代码段
01
10104H
00
10103H
05
指令2
CS
10102H
00
10101H
64
10100H
B8
指令1
低地址
播放
停止

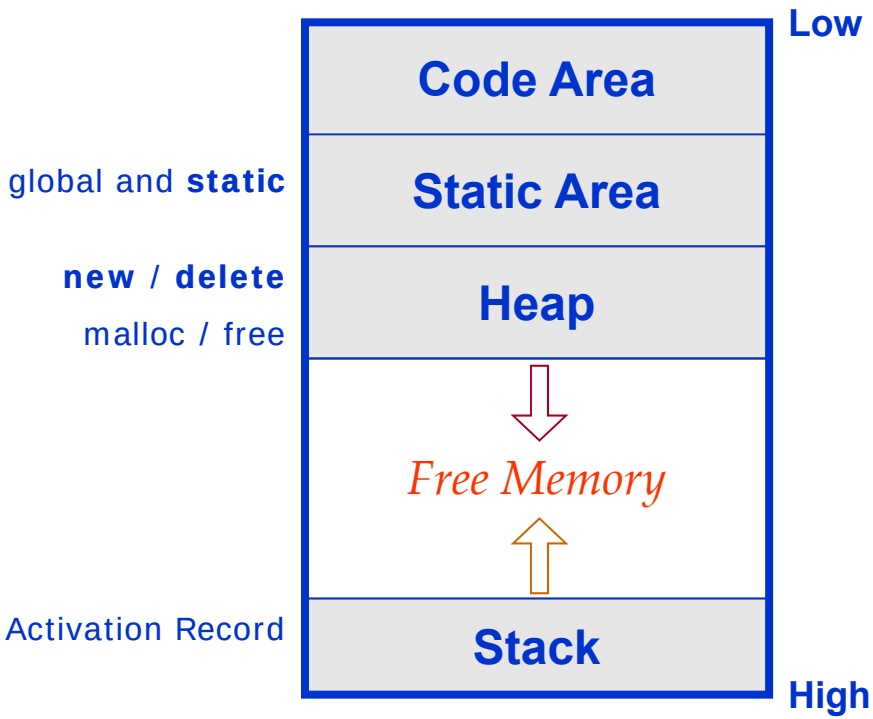
Why is C++

编程语言	表示形式
C 语言	a=b+1;

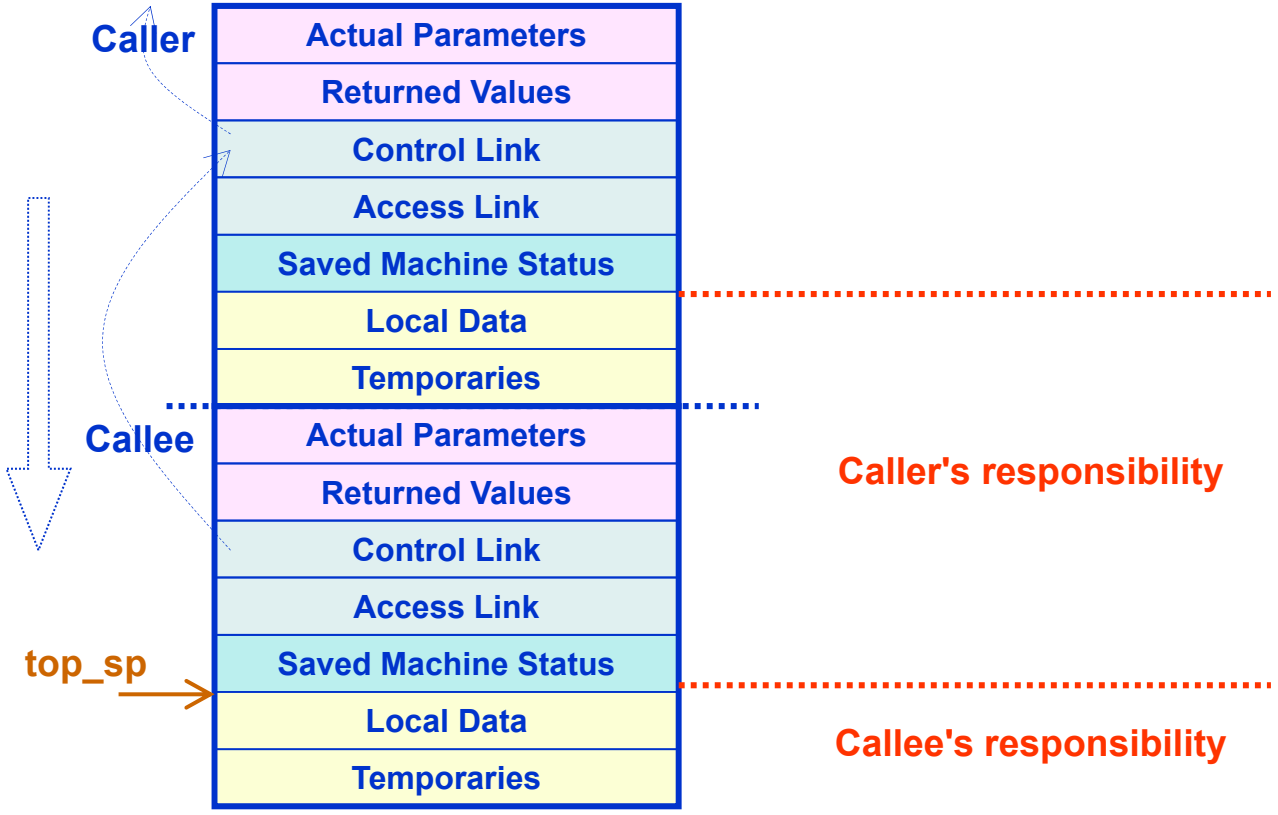
汇编语言	mov 0x804a01c,%eax
	add $0x1,%eax
	mov %eax,0x804a018

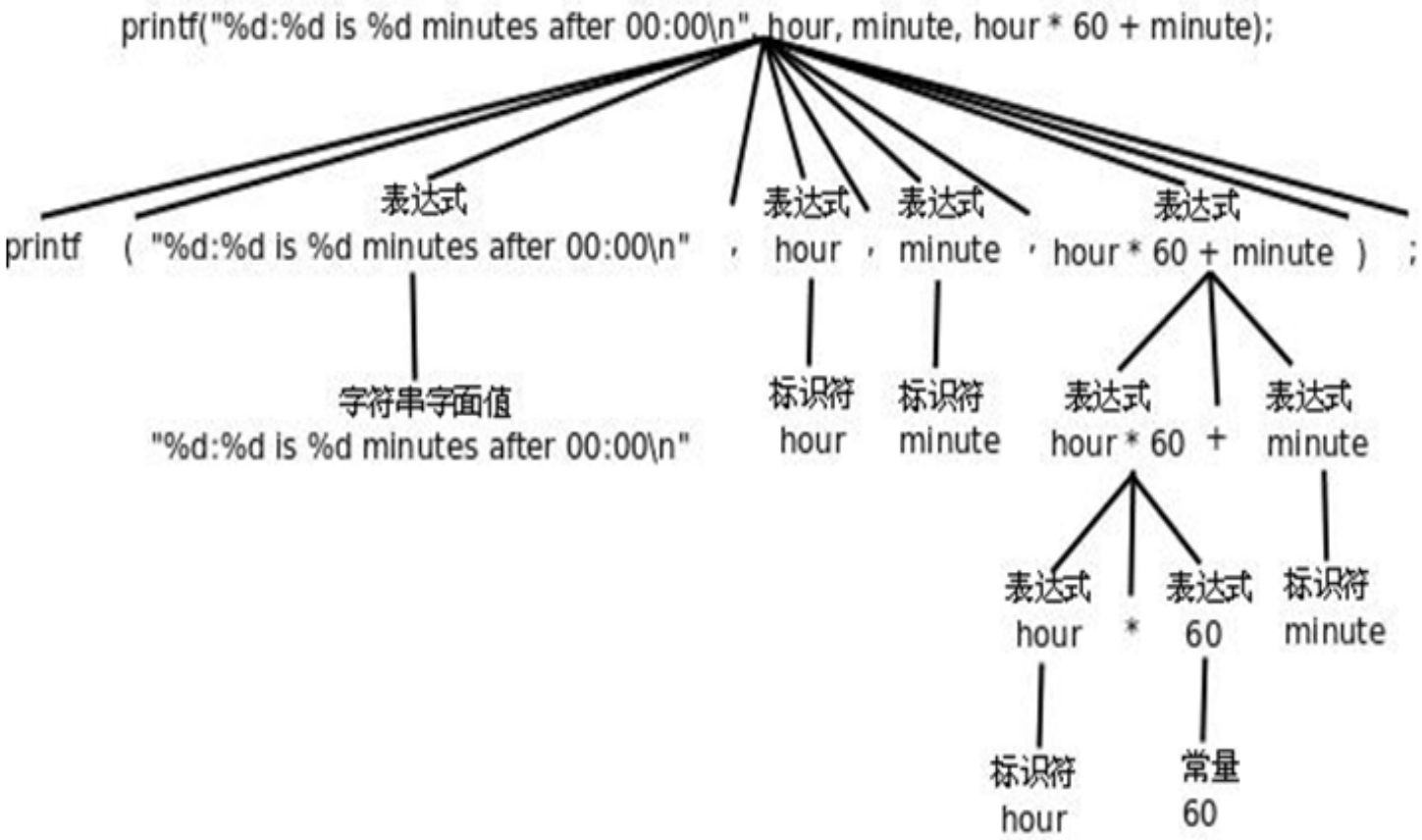
机器语言	a1 1c a0 04 08 83 c0 01 a3 18 a0 04 08

Why is C++


Low
Code Area
global and static
Static Area
new / delete
malloc / free
Heap
Free Memory
Activation Record
Stack
High

Why is C++


Caller
Actual Parameters
Returned Values
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries
Callee
Actual Parameters
Returned Values
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries
top_sp
Caller's responsibility
Callee's responsibility


printf("%d:%d is %d minutes after 00:00\n", hour, minute, hour * 60 + minute);
printf ("%d:%d is %d minutes after 00:00\n" , hour , minute , hour * 60 + minute) ;
表达式
字符串字面值
"%d:%d is %d minutes after 00:00\n"
表达式
标识符
hour
表达式
标识符
minute
表达式
hour * 60 + minute
表达式
hour * 60
表达式
minute
表达式
hour
表达式
60
标识符
minute
标识符
hour
常量
60

表达式 → 标识符

表达式 → 常量

表达式 → 字符串字面值

表达式 → (表达式)

表达式 → 表达式 + 表达式

表达式 → 表达式 - 表达式

表达式 → 表达式 * 表达式

表达式 → 表达式 / 表达式

表达式 → 表达式 = 表达式

语句 → 表达式;

语句 → printf(表达式, 表达式, 表达式, ...);

变量声明 → 类型 标识符 = Initializer, 标识符 = Initializer, ...;（= Initializer的部分可以不

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*.

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++

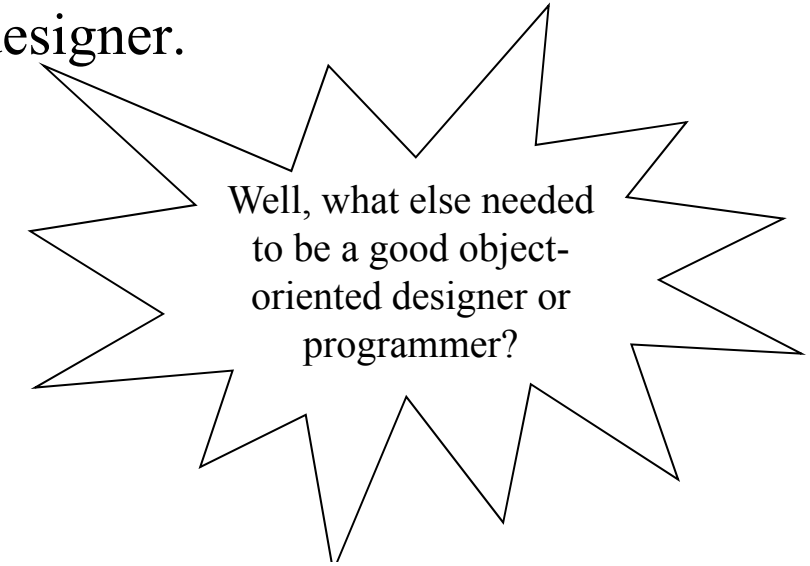
- The most important thing to do when learning C++ is to *focus on concepts* and not get lost in *language-technical details*.

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.



面向过程设计（ procedural programming ）

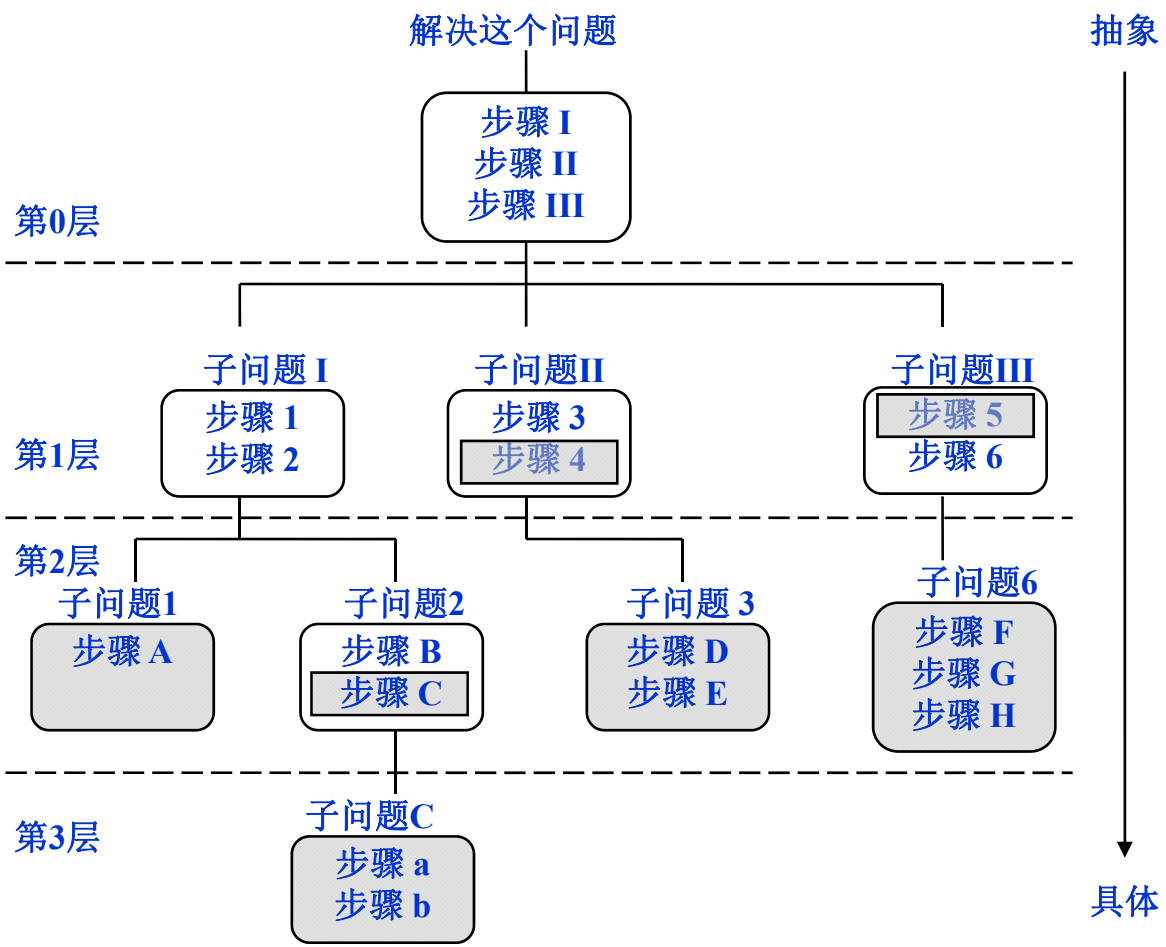
- 也称为：

1. 功能分解（Function Decomposition）
2. 自上而下逐步求精（top-down design, stepwise refinement）。

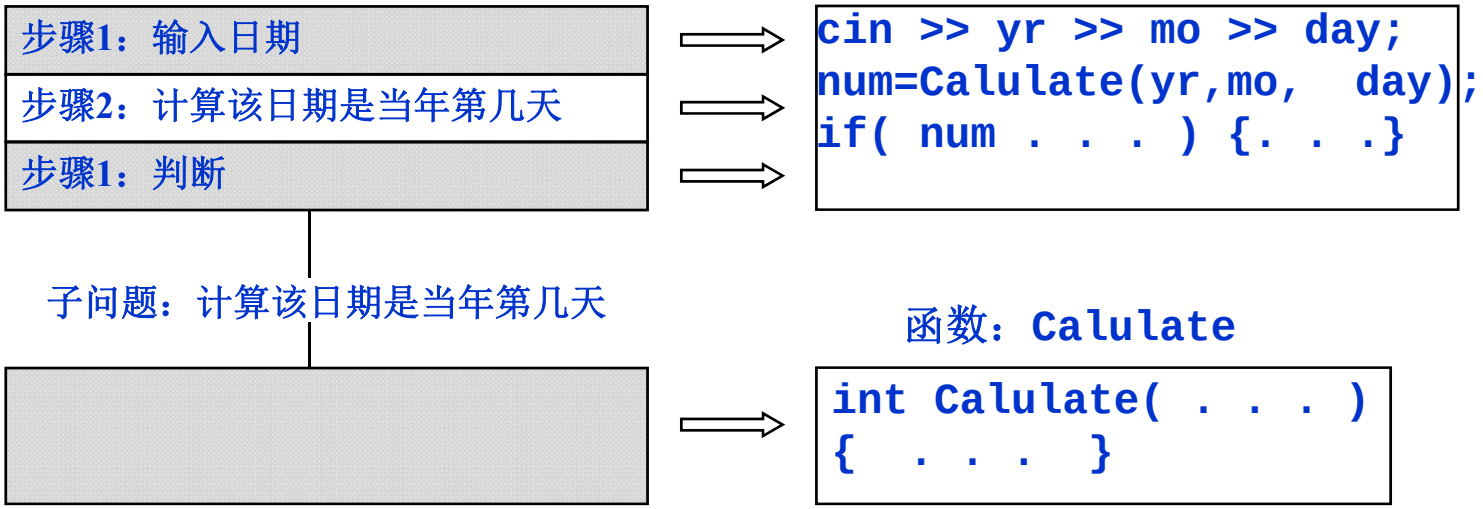
面向过程设计（procedural programming）

- 围绕功能设计程序：采用自顶向下分解的方法分析问题域所要实现的功能，使用算法描述实现功能的步骤。
- 对于复杂功能则进行分解以使得每个功能都可设计算法实现，然后将这些算法组合起来实现该复杂功能。
- 结构化程序设计的主要工作就是功能分解和算法设计。

面向过程设计（procedural programming）



面向过程设计（ procedural programming ）



面向过程设计（procedural programming）

- 利用这种方式思考问题时，我们把问题看成是什么？
- 为什么在解决某个问题时，有一些步骤仍为抽象？为什么不一次性把所有步骤的细节全部写出来？

面向过程设计（procedural programming）

- 在构建大型软件系统时，面向过程的设计往往导致程序有两大致命缺陷：
  - 导致程序结构不灵活。若高层算法需要修改，那么可能底层的算法也因此需要修改。
  - 导致代码难以复用。
  - 其实，面向过程设计的最大问题是。。。。。

很多问题不是步骤序列

面向对象程序设计

- 面向对象程序设计
  - 是一种理念（idea）：思维和方法论的问题。
  - 是某种语言里面支持面对对象的具体机制：程序语言的运用问题

----（类和对象、继承、多态性和模板等）。

Abstract Data Type（抽象数据类型, ADT）

- 在程序设计中，对于被抽象的数据，称为抽象数据类型（Abstract Data Type, ADT）。
- 一种ADT应具有

1 说明部分（说明该该ADT是什么及如何使用）：说明部分描述数据值的特性和作用于这些数据之上的操作。ADT的用户仅须明白这些说明，而无须知晓其内部实现。

2 实现部分。

Abstraction

- 用户：抽象是用户的“权利”。
- 设计者：必须关注内部实现。
- 被抽象的对象本身：必须具备说明部分用以向用户说明自身，以及具备具体的实现部分。

An ADT: DATE

ADT Implementation means

1） Choosing a specific **data representation** for the abstract data using data types that already exist (built-in or programmer-defined).

2）**Writing functions** for each allowable operation.

Type

DATE

Data

Each DATE value is date in day/month/year → int year, month, day;

Operation

Set the date

Get the date

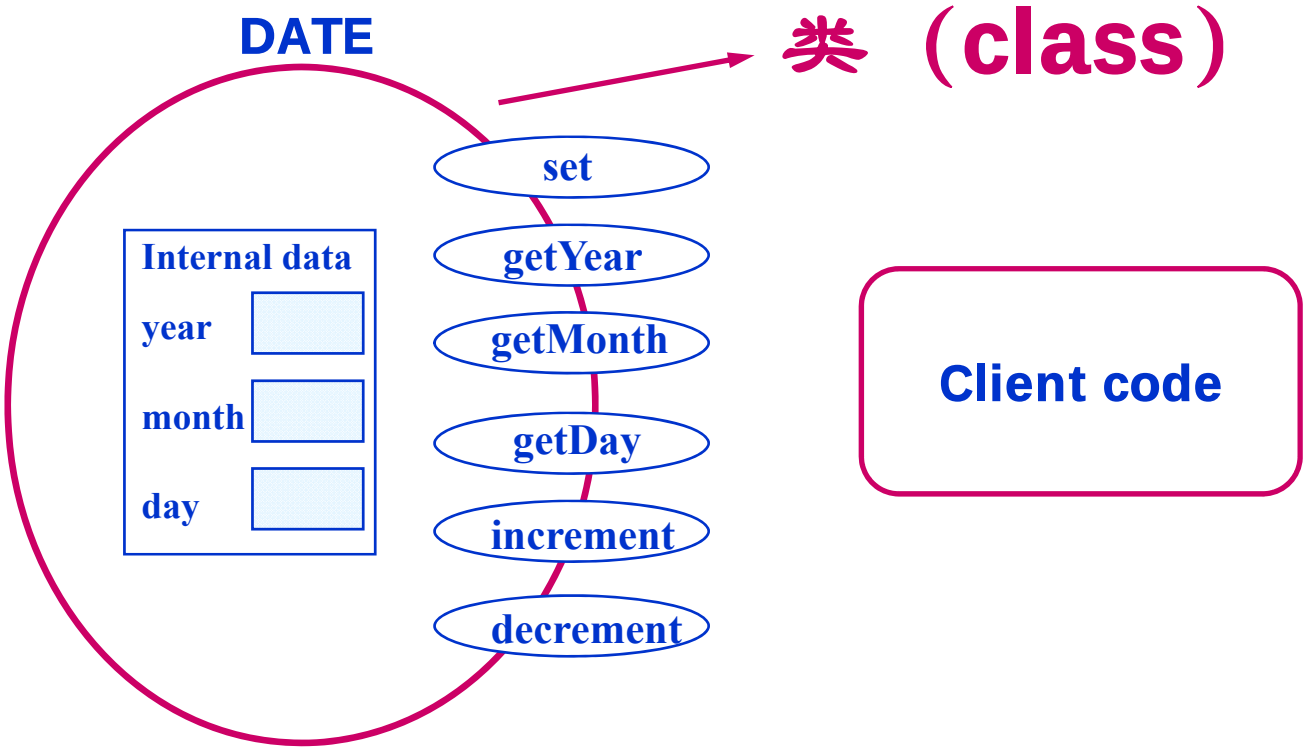
Increment the date by one day → set() get() increment()

Decrement the date by one day → decrement()

更深入的思考。。。

- 把DATE设计为一种数据类型。
- 内部包含年月日等数据以及在这些数据上可进行的操作。
- 用户利用DATE就可以定义多个变量。
- 用户可调用每个变量中公开的操作，但无法直接访问每个变量中被隐藏的内部数据。
- 用户也无需关心变量中各操作的具体实现。
- 于是DATE就是一种封装好的数据类型。这就达到了信息隐藏和封装的目的。

Implementing DATE with a class



Class and Object

- Class: is a user-defined data type that represents an ADT in C++ that have attributes (data members) and behaviors (member functions) that operate on the data members.
- Variables of the class type are called class objects(对象) or class instances(实例).
- Software that uses the class is called a client(客户代码). Client code uses public member functions to handle its class objects.

Implementing DATE with a class

```
class DATE   // DATE.h----Specification file of class DATE
{
    public:
        void Set( int, int, int );
        int getMonth() const;
        int getDay() const;
        int getYear() const;
        void Print() const;
        void Increment();
        void Decrement();
    private:
        int month;
        int day;
        int year;
};
```

Implementing DATE with a class

- class是保留字，说明DATE是类名。在{}中列出类的成员。
- 类的成员包括：
- 数据成员：一般说来，数据成员是需要隐藏的对象；即外部的程序是不能直接访问这些数据的，应该通过函数成员来访问这些数据。所以一般情况下，数据成员通过关键字private声明为私有成员（private member）
- 函数成员：通过关键字public声明为公有成员（public member）。外部程序可以访问共有成员，但无法访问私有成员。
- 对于类的使用者（即用户代码，简称用户）而言，只需要获得DATE.h，即可调用类对象的公有函数访问其内部的数据成员。使用者无法直接访问私有成员，也无需知晓公有函数的内部实现。

Implementing DATE with a class

```
//DATE.cpp
//the implementation of
each member function of
DATE.

#include "DATE.h"
#include <iostream>
using namespace std;

int DaysInMonth( int, int );

void DATE::Set(int newYear,
                    int
newMonth,
                    int newDay )
{  }
```

```
int DATE::getMonth() const
{  }
int DATE::getDay() const
{  }
int DATE::getYear() const
{  }
void DATE::Print() const
{  }
void DATE::Increment()
{  }
void DATE::Decrement()
{  }
int DaysInMonth(  int mo,
                        int yr  )
{  }
```

Implementing DATE with a class

```
//DATE.cpp  the implementation of each member
//function of DATE.

void DATE::Set(int newYear,
               int newMonth,
               int newDay )
{
   month = newMonth;
   day = newDay;
   year = newYear;
}
```

Implementing DATE with a class

```
//DATE.cpp  the implementation of each member function of
//DATE.

int DATE::getMonth() const
{
   return month;
}
int DATE::getDay() const
{
   return day;
}

int DATE::getYear() const
{
   return year;
}
```

Implementing DATE with a class

```
//DATE.cpp  the implementation of each member function
of //DATE.
void DATE::Print() const
{
   switch (month)
   {
      case 1 : cout << "January";
               break;
      case 2 : cout << "February";
               break;
                  :
      case 12 : cout << "December";
   }
   cout << ' ' << day << ", " << year << endl << endl;
}
```

Implementing DATE with a class

```
//DATE.cpp  the implementation of each member function of
//DATE.
void DATE::Increment()
{
   day++;
   if (day > DaysInMonth(month, year))
   {
      day = 1;
      month++;
      if (month > 12)
      {
         month = 1;
         year++;
      }
   }
}
```

Implementing DATE with a class

```
//DATE.cpp  the implementation of each member function of DATE.
void DATE::Decrement()
{  day--;
   if ( day == 0 )
   {
      if( month == 1 )
      {
         day = 31;
         month = 12;
         year--;
      }
      else
      {
         month--;
         day = DaysInMonth( month, year );
      }
   }
}
```

Implementing DATE with a class

```
//DATE.cpp  the implementation of the auxiliary function
//DaysInMonth.
int DaysInMonth( /* in */ int mo, /* in */ int yr  )
{
   switch (mo)
   {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
           return 31;
    case 4: case 6: case 9: case 11:
          return 30;
    case 2:
        if ((yr % 4 == 0 && yr % 100 != 0) ||yr % 400 == 0)
           return 29;
        else
           return 28;
   }
}
```

Implementing DATE with a class

- 在DATE.cpp文件开头需要加入预处理命令

  #include “DATE.h”

  这是因为在DATE.cpp中要用到用户自定义的标识符DATE，而它的定义在DATE.h中。

- 在DATE.h中，各函数原型是在{}中的。根据标识符的作用域规则，它们的作用范围仅在类定义中，而不包括DATE.cpp。因此在DATE.cpp中需要利用作用域解释运算符“::”来指明这里的函数是类DATE里的成员函数。

- DATE.cpp中有时还包括DATE内部要使用到的函数，例如DaysInMonth。这种函数并非对外公开供用户使用，因此可以将其声明为类的私有成员。若在该函数中没有涉及该类的数据成员，则无需将它们声明为类的成员。

Client Code Using DATE

```
//client.cpp
#include "DATE.h"
#include <iostream>
using namespace std;
int main()
{
    DATE date1, date2; //①
    int tmp;

    date1.Set( 1989, 6, 4 );
    date1.Print();
    date1.Increment();
    date1.Print();
                :
}
```

```
    date2.Set( 1997, 7, 1 );
    date2.Print();
    date2.Decrement();
    date2.Print();

    tmp = date1.getYear();
    tmp++;
    date1.Set( tmp, 12, 20 );
    date1.Print();

    cout << date1.year;
                        //error?

    return 0;
}
```

Results

The following items will be displayed on the monitor.

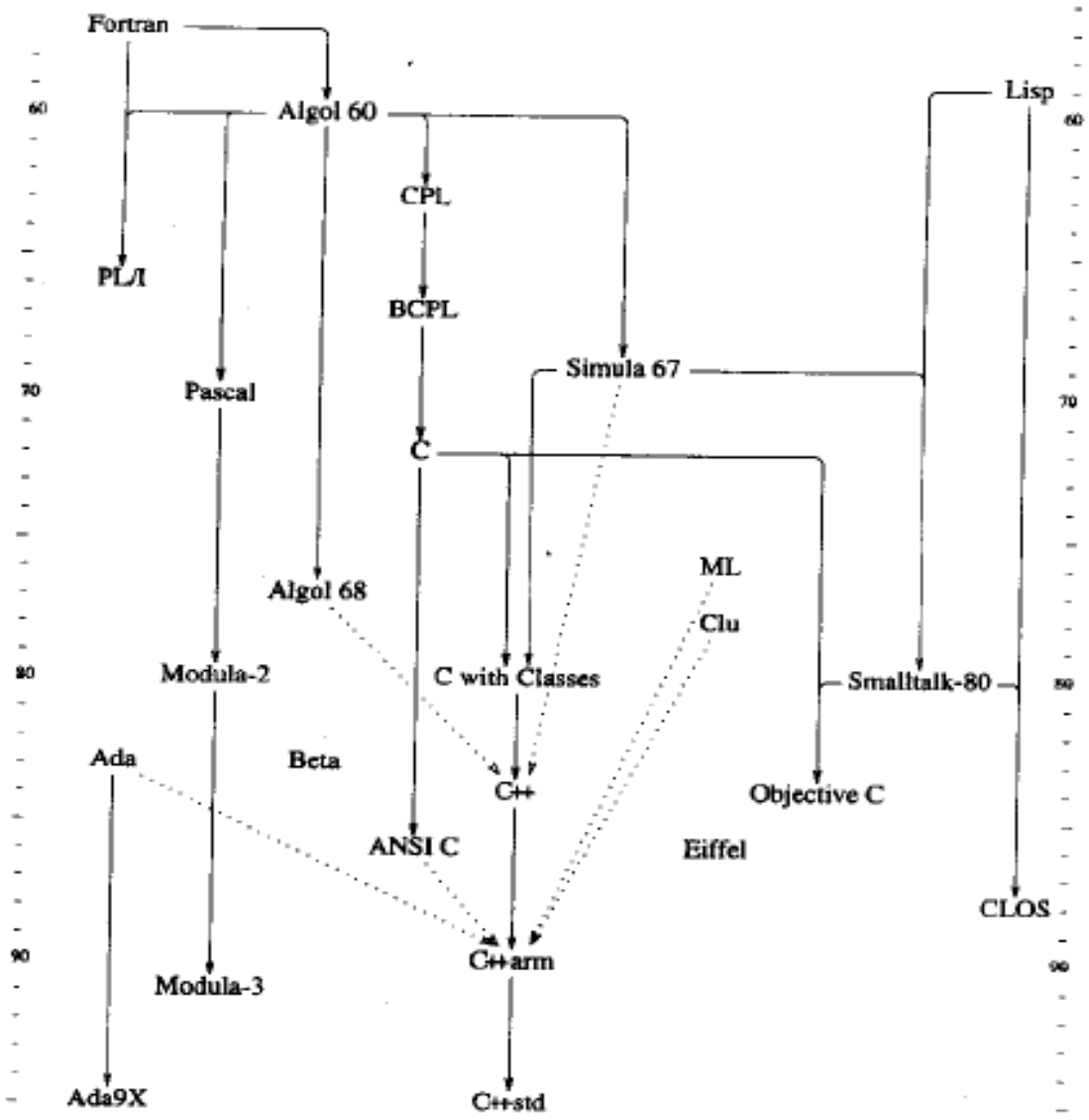
```
June 4, 1989
June 5, 1989
July 1, 1997
June 30, 1997
December 20, 1990
```

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


Fortran
Algol 60
Lisp
CPL
PL/I
BCPL
Simula 67
Pascal
C
ML
Algol 68
Clu
Modula-2
C with Classes
Smalltalk-80
Ada
Beta
C++
Objective C
ANSI C
Eiffel
CLOS
C++arm
Modula-3
Ada9X
C++std

Development platforms and compliers

languages	Development platform	Compiler
C	Turbo C Visual C++ C++Builder	集成在开发平台中
C++	Turbo C Visual C++ C++Builder	同上
Basic	Visual Basic	同上
Java	Visual J++ J++Builder JCreator	JDK，集成在开发环境中

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
  - 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;
}
```


1-1.cpp

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

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

Namespace

- *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.
- The scope of an identifier declared in a namespace definition extends from the point of declaration to the end of the namespace body, and its scope includes the scope of a using directive specifying that namespace.

Namespace

- Suppose the codes in header file iostream are as follows.

```
namespace std
{
   istream cin;
   ostream cout;
}
```

- What does it mean if we add the following statement in the .cpp file?

```
#include< iostream >
```

Namespace

```
#include< iostream >
main()
{
    int a;
    cin >>  a;
    cout << a ;
    return 0;
}
```

After being processed by the preprocessor, the code will be extended as…

Namespace

```
namespace std
{
    istream cin;
    ostream cout;
}
int main()
{
    int a;
    cin >>  a;
    cout << a ;
    return 0;
}
```

经预处理器处理后，扩展为左边的代码。根据前面所述的标识符作用范围规则：**cin**和**cout**是在名字空间**std**内声明的，而**std**是一个块（**block**），因此在其外当然不能直接使用**cin**和**cout**。所以左边代码实际上有语法错误。

因此才出现

using namespace std;

这样的语句来解决这个问题。

3 Ways to Use Namespace Identifiers

- use a qualified name consisting of the namespace, the scope resolution operator :: and the desired the identifier

```
std::cin >>  a;
```

- write a using declaration

```
using  std::abs ;
cin >>  a;
```

- write a using directive locally or globally

```
using  namespace  std ;
cin >>  a;
```

//看程序namespace

Namespaces

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

Namespaces

```
namespace mfc {  //vendor 1's namespace
      int inflag;     //vendor 1's inflag
}
namespace owl {  //vendor 2's namespace
      int inflag;    //vendor 2's inflag
}
using namespace mfc;
using namespace owl;

int main()
{
              :
   inflag = 3;
              :
}
```

//看程序namespace1

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(8);                //Error!
mfc::g(8);           //OK, full name
using mfc::g;        //using declaration for g
g(8);                //OK
```

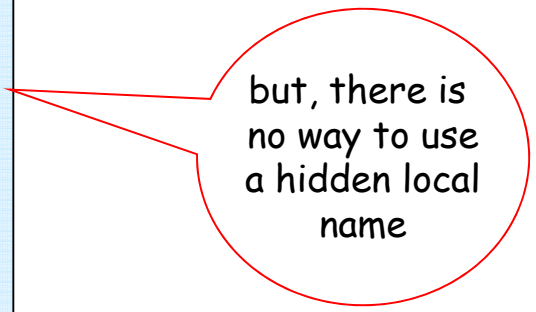


```
namespace mfc {  //vendor 1's namespace
        int inflag;   //vendor 1's inflag
        void g(int);
}
using namespace mfc;   //using directive
inflag = 21;           //mfc::inflag
g(-66);                //mfc::g
owl::inflag=341;       //full name needed
```

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
}
```



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*

I/O

- No I/O is built into C++.
- Instead, a library provides input/output streams for I/O.

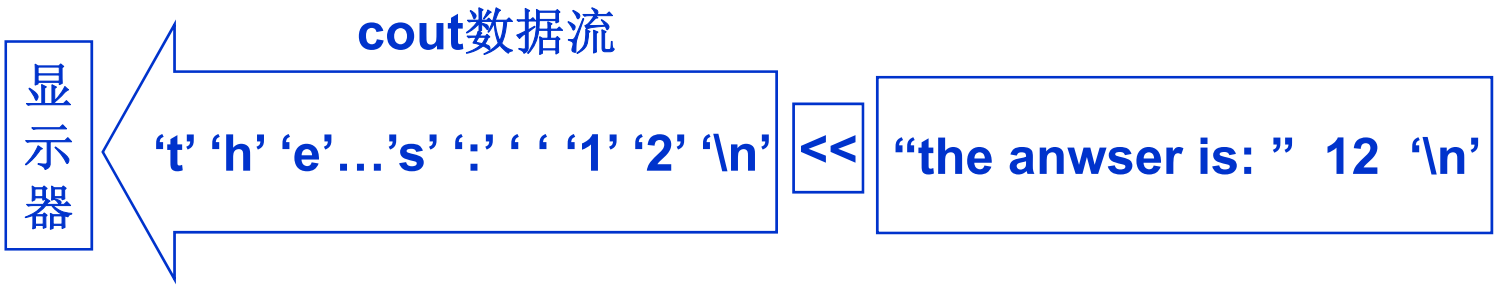
I/O

```
#include <iostream>  //看程序IO
using namespace std;
int main()
{
  int someInt;
  float someFloat;
  char someChar;

  cout << "the answer is: " << 3*4 << endl;
  cin  >> someInt  >> someFloat >> someChar ;
  return 0;
}
```

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
```

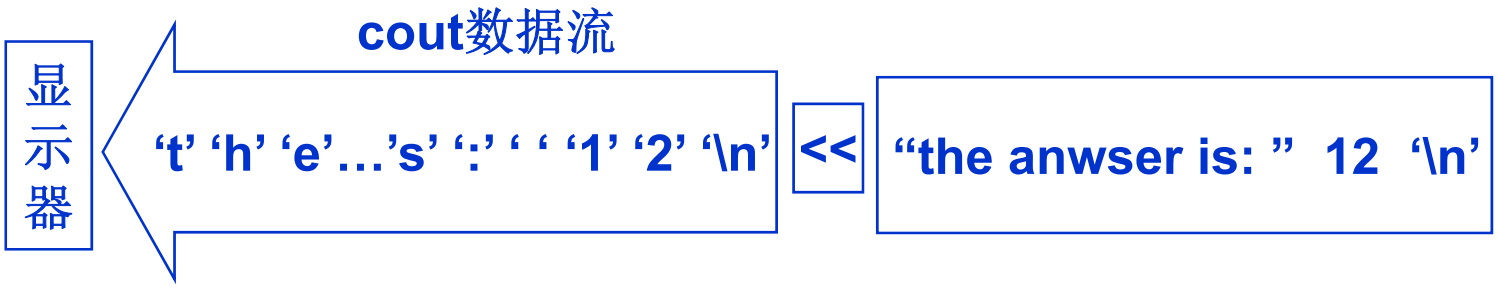
1.计算机对3*4求值得整数值12；
2. <<把字符‘t’、’ h’ … ‘s’、’ :’、’ ‘放入cout流中；
3. <<把整数值12转化为字符‘1’和‘2’，也放入cout流中
4. endl产生一个换行符，该字符也被放入cout流中
5.cout把这些字符送往显示器



cout与输出的实质

```
cout << “the answer is: ” << 3*4 << endl;
```

```
cout << “the answer is: ”;
cout << 3*4;
cout << endl;
```



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

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

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

‘ ’ ‘9’ >> someChar

memory

功能：读取、转化

Manipulators

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

Manipulators: Fixed and Showpoint

- use the following statement to specify that (for output sent to the cout stream) decimal format (not scientific notation) be used, and that a decimal point be included (even for floating values with 0 as fractional part)

```
cout << fixed << showpoint ;
```

Manipulators: setprecision(n)

- requires #include <iomanip> and appears in an expression using insertion operator (<<)
- if fixed has already been specified, argument n determines the number of places displayed after the decimal point for floating point values
- remains in effect until explicitly changed by another call to setprecision

What is exact output?

```
#include  <iomanip > // for setw( ) and setprecision( )
#include  <iostream>
using  namespace  std;

int main (  )
{
   float   myNumber  =  123.4587 ;
   cout  <<  fixed  <<   showpoint  ;          // use decimal format
                                              // print decimal points
   cout  <<  "Number is "  <<  setprecision ( 3 )
         <<  myNumber     <<  endl ;

   return  0 ;
}
```

Number is 123.459

Manipulator: setw

- "set width" lets us control how many character positions the next data item should occupy when it is output
- setw is only for formatting numbers and strings, not char type data

setw(n)

- Requires #include <iomanip> and appears in an expression using insertion operator (<<).
- Argument n is called the fieldwidth specification, and determines the number of character positions in which to display a right-justified number or string (not char data). The number of positions used is expanded if n is too narrow.
- “set width” affects only the very next item displayed, and is useful to align columns of output .

What is exact output?

```
#include  <iomanip>                    // for setw( )
#include  <iostream>
#include  <string>

using  namespace  std;

int  main ( )
{
   int  myNumber    =  123 ;
   int  yourNumber  =  5 ;

   cout  <<  setw ( 10 )    <<   "Mine"
         <<   setw ( 10 )    <<  "Yours"          <<   endl;
          <<   setw ( 10 )    <<  myNumber
         <<   setw ( 10 )    <<  yourNumber   <<   endl ;

   return 0 ;
}
```

OUTPUT

position 12345678901234567890

```
      Mine      Yours
       123          5
```

each is displayed right-justified and
each is located in a total of 10 positions

What is exact output?

```
#include  <iomanip>                    // for setw( ) and setprecision( )
#include  <iostream>

using  namespace  std;

int  main ( )
{
   float myNumber    =  123.4 ;
   float yourNumber  =  3.14159 ;

   cout  <<  fixed  <<   showpoint  ;    // use decimal format
                                         // print decimal points

   cout  <<  "Numbers are: "  <<  setprecision ( 4 ) <<  endl
         <<   setw ( 10 )          <<  myNumber     <<   endl
         <<   setw ( 10 )          <<  yourNumber   <<   endl ;

   return 0 ;
}
```

OUTPUT

```
12345678901234567890
Numbers are:
   123.4000
     3.1416
```

each is displayed right-justified and rounded if necessary and each is located in a total of 10 positions with 4 places after the decimal point

Manipulators

```
#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

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;
}
```

lo.cpp

```
#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

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 *outstream* 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);
        }
 ……
```

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

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);
```

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;

        cout << i << endl << j <<
                endl << *p << endl << *q <<endl;
        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.

Casts

- 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 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;
```

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

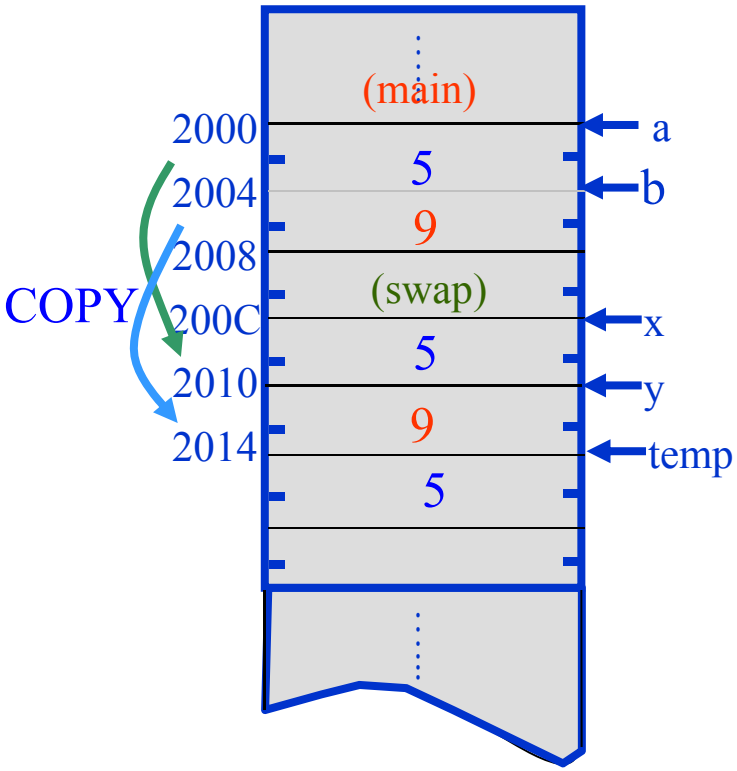
```
int idx = s1.find( s2 );
if( idx < s1.length() )
        cout << "Found at index: "<< idx << endl;
else
        cout << "Not found" << endl;
if( s1 > s2 )
        cout <<"\""+s1+"\""<<
        " is greater than "<<"\""+s2+"\""<<endl;
else
        cout <<"\""+s1+"\""<<
        " is not greater than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

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

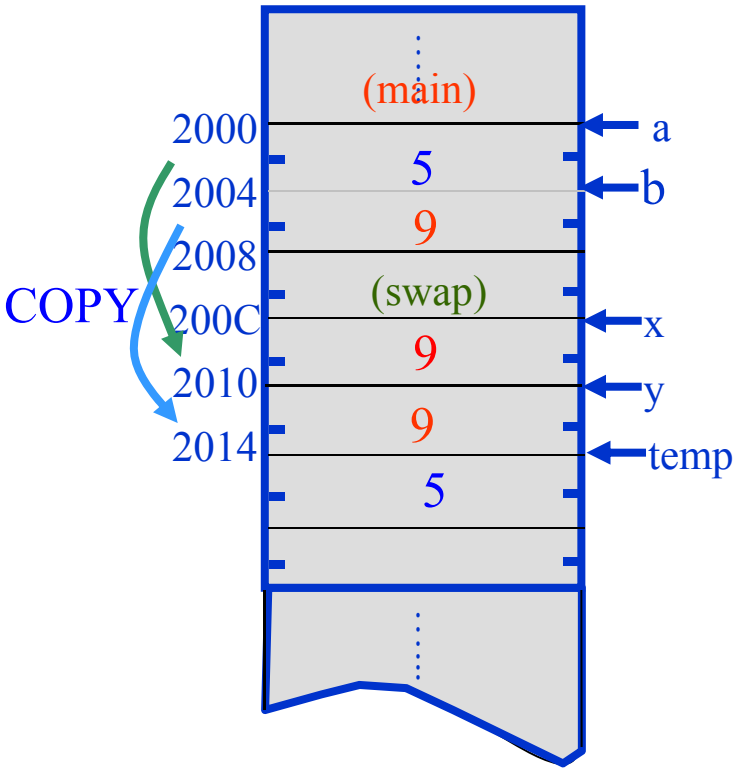
```
#include <stdio.h>
void swap(int  x,int y)
{
      int  temp;
      temp=x;
      x=y;
      y=temp;
}
int main()
{
      int a,b;
      scanf("%d,%d",&a,&b);
      swap(a,b);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```



Swap1.c

Call by Value

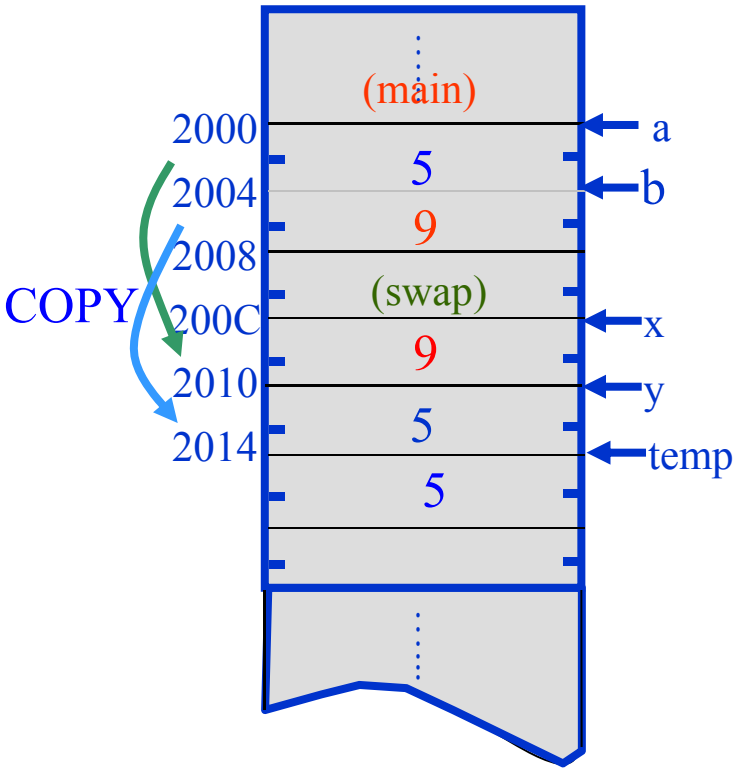
```
#include <stdio.h>
void swap(int  x,int y)
{
      int  temp;
      temp=x;
      x=y;
      y=temp;
}
int main()
{
      int a,b;
      scanf("%d,%d",&a,&b);
      swap(a,b);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```



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;
}
```

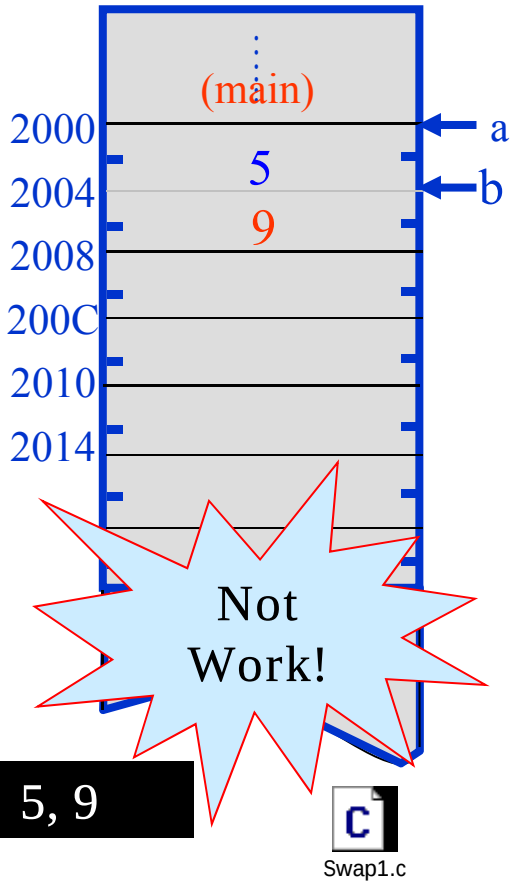


Swap1.c

Call by Value

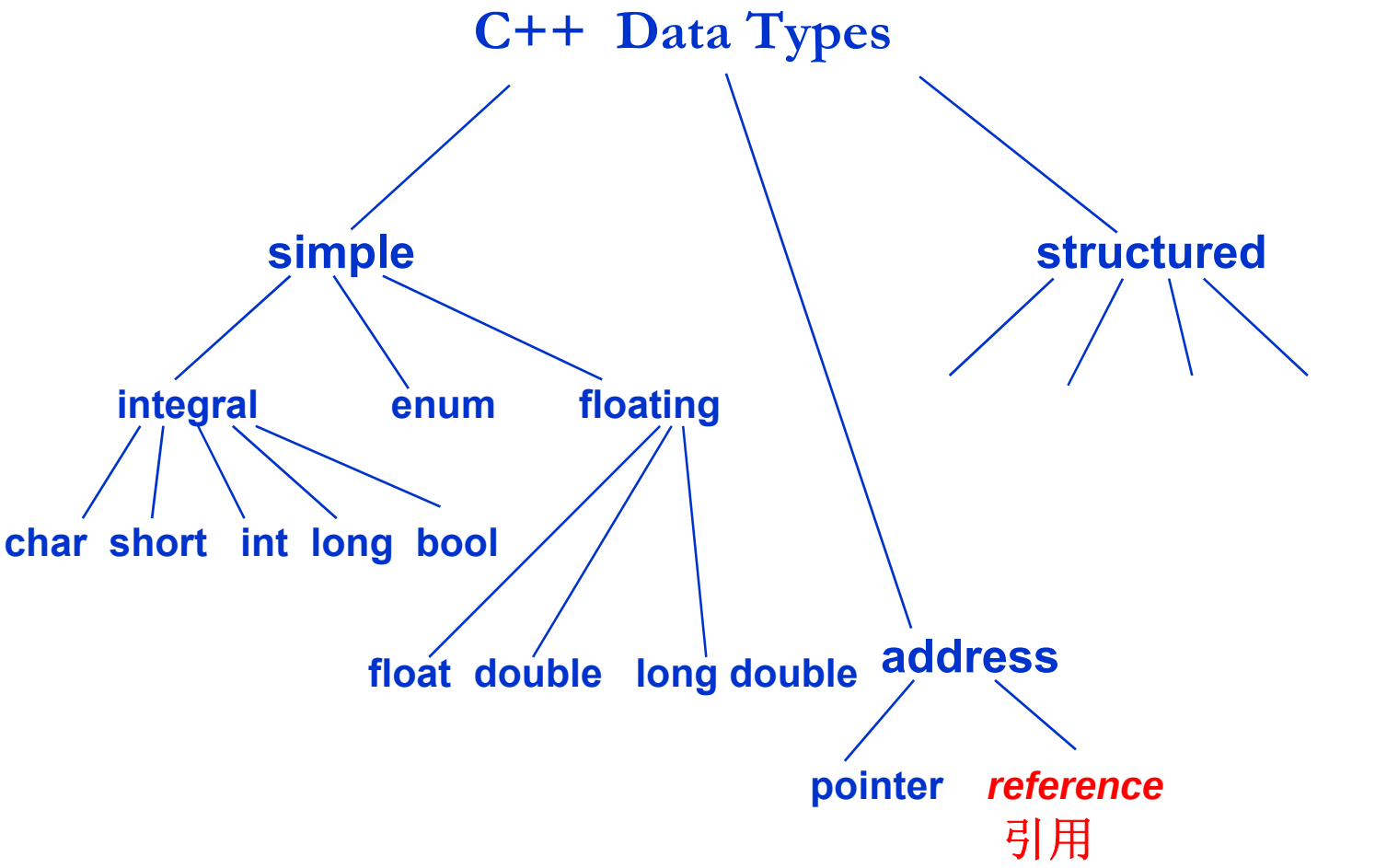
```
#include <stdio.h>
void swap(int  x,int y)
{
      int  temp;
      temp=x;
      x=y;
      y=temp;
}
int main()
{
      int a,b;
      scanf("%d,%d",&a,&b);
      swap(a,b);
      printf("\n%d,%d\n",a,b);
      return 0;
}
```

Pass by value



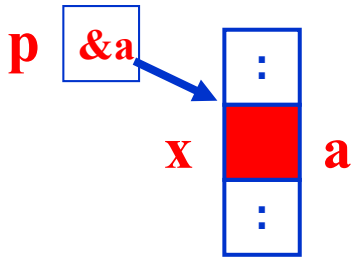
Result: 5, 9

Swap1.c


C++ Data Types
simple
structured
integral
enum
floating
char
short
int
long
bool
float
double
long double
address
pointer
reference
引用

例1 利用引用为变量再起名称

```
int main( )   //程序Ref1
{
   int a = 1024;
   int *p  = &a;      // p是指针; &a是a的地址
   int& x = a;        // x是引用，它实际上与a是同一个变量

   cout << “a = " << a << endl;
   cout << “x = " << x << endl;
   cout << “ *p = " << *p << endl;

   x = 2000;
   cout << “a = ” << a << endl;

   return 0;
}
```

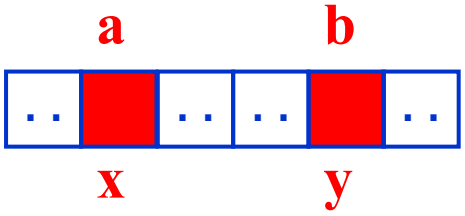



```
a = 1024
x = 1024
*p = 1024
a = 2000
```

例2：在形参中使用引用

```
//程序RefSwap

void swap( int& x, int& y )
{
   int temp;

   temp = x;
   x = y;
   y = temp;
}
```

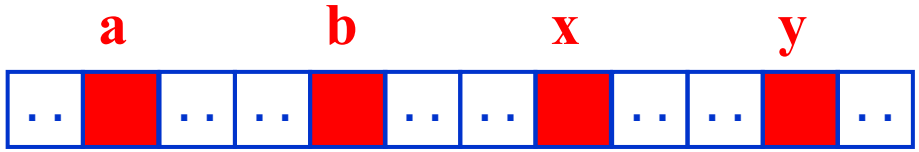
```
int main()
{
   int a, b;
   cin >> a >> b;

   swap( a, b );

   cout << "max = " << a
        << " "
        << "min = " << b;
}
```



假如形参不使用引用

```
void swap( int x, int y )
{
   int temp;

   temp = x;
   x = y;
   y = temp;
}
```

```
int main()
{
   int a, b;
   cin >> a >> b;

   swap( a, b );

   cout << "max = " << a
        << " "
        << "min = " << b;
}
```

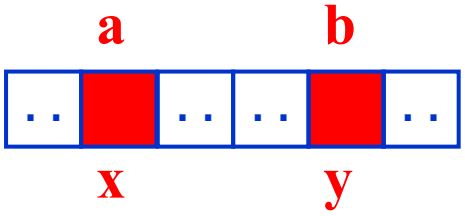


函数调用时，a的值传递给x，b的值传递给y。
接着，a与x、b与y再无任何关系。

例2：在形参中使用引用

```
//程序RefSwap

void swap( int& x, int& y )
{
   int temp;

   temp = x;
   x = y;
   y = temp;
}
```

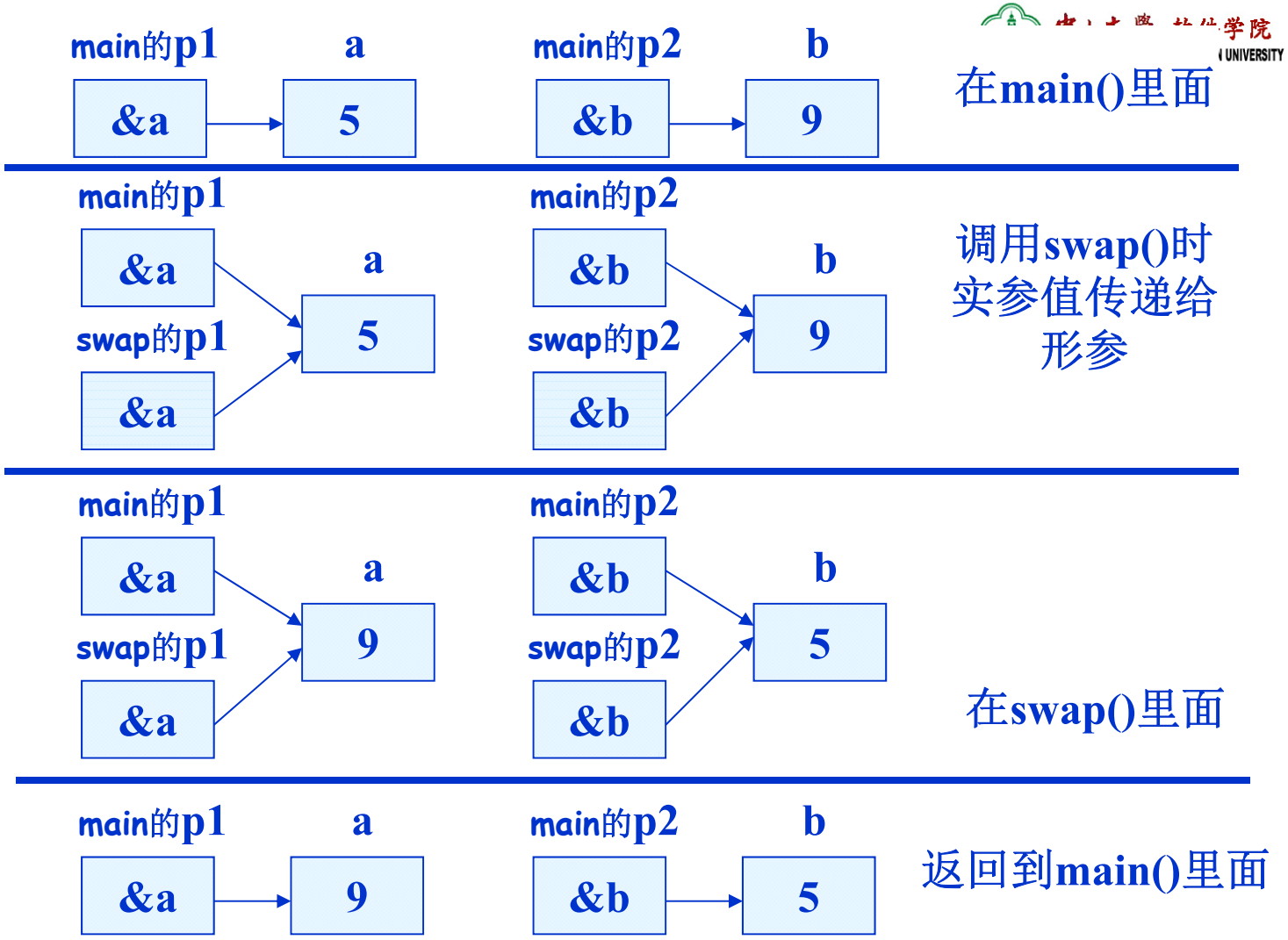
```
int main()
{
   int a, b;
   cin >> a >> b;

   swap( a, b );

   cout << "max = " << a
        << " "
        << "min = " << b;
}
```



与引用相比，这个指针例子又如何？

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

   temp = *p1;
   *p1 = *p2;
   *p2 = temp;
}
```

```
int main()
{
    int a =5 , b =9;
    int *p1, *p2;

    p1 = &a;
    p2 = &b;

    swap( p1, p2 );

    cout << “a = " << a << “ ”
         << “b = " << b;
}
```


main的p1
a
&a
5
main的p2
b
&b
9
在main()里面
main的p1
&a
a
5
swap的p1
&a
main的p2
&b
b
9
swap的p2
&b
调用swap()时
实参值传递给
形参
main的p1
&a
a
9
swap的p1
&a
main的p2
&b
b
5
swap的p2
&b
在swap()里面
main的p1
a
&a
9
main的p2
b
&b
5
返回到main()里面

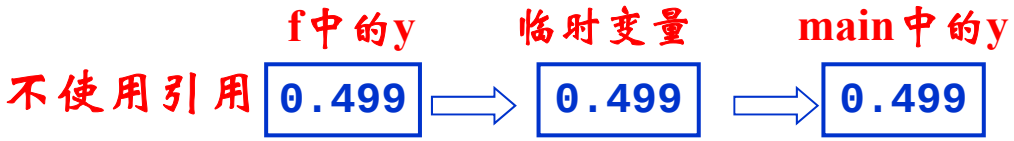
指针与引用

- 相同：可以使一个函数向调用者返回多个数值。
- 不同：原理不同。引用传递中，形参、实参实质为同一变量，或者说是为某个变量起多了一个名字。而使用指针作函数参数，则是使被调用函数获得某变量的地址，从而使用这个地址访问这个变量。
- 从返回值的角度看，引用形参比利用指针方便。

例3 返回值不使用引用

```
double f( double x )
{
   double y;

   y = sin(x);

   return y;
}
```

```
int main()
{
   double a = 3.14/6;
   double y;

   y = f( a );

   cout << "y = " << y << endl;

   return 0;
}
```



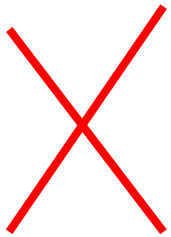
例3 返回值使用引用

```
double& f( double x )     //程序ref2
{
   static double y;

   y = sin(x);

   return y;
}
```

```
int main()
{
   double a = 3.14/6;
   double y;

   y = f( a );

   cout << "y = " << y << endl;

   return 0;
}
```

例3 返回值使用引用

```
double& f( double x )
{
   double y;

   y = sin(x);

   return y;
}
```



```
int main()
{
   double a = 3.14/6;
   double y;

   y = f( a );

   cout << "y = " << y << endl;

   return 0;
}
```

Inline Function

```
#include <iostream>
using namespace std;
inline void swap(int&, int&);
int main() {
        int i=7, 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;
}
```

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.

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

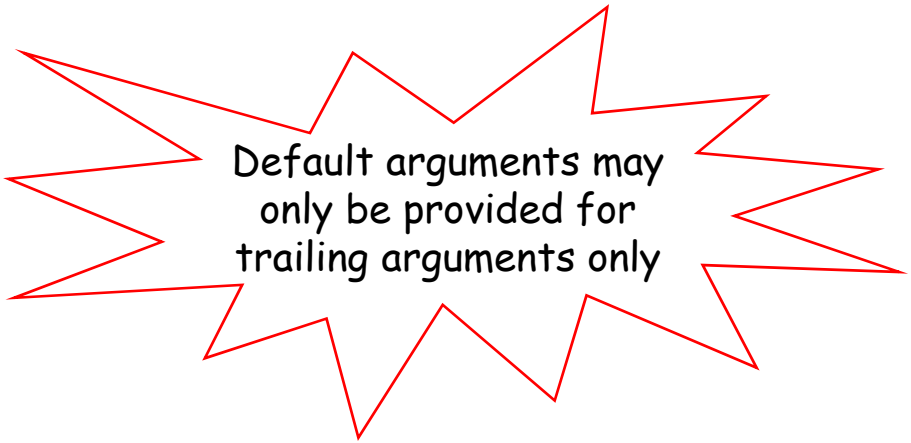
- Dissimilarities
  - A macro is expanded by the *preprocessor*, an inline function is expanded by the *compiler*.
  - Macro expansions do text substitution *without regard to the semantics* of the code; but inline function expansions *take into account the semantics*.
    - Macro:No type-safety checking.
    - 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" );
```



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);
```

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

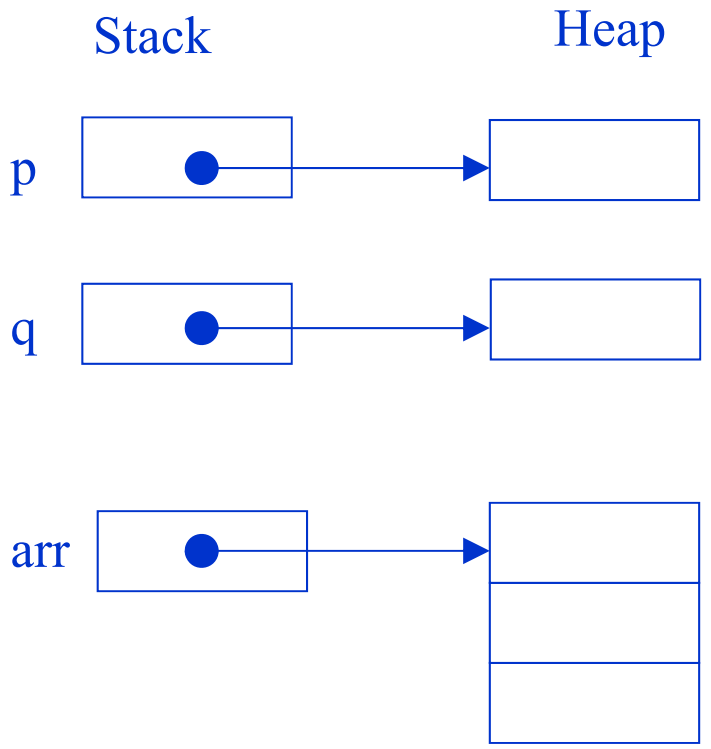
The new And delete Operators

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

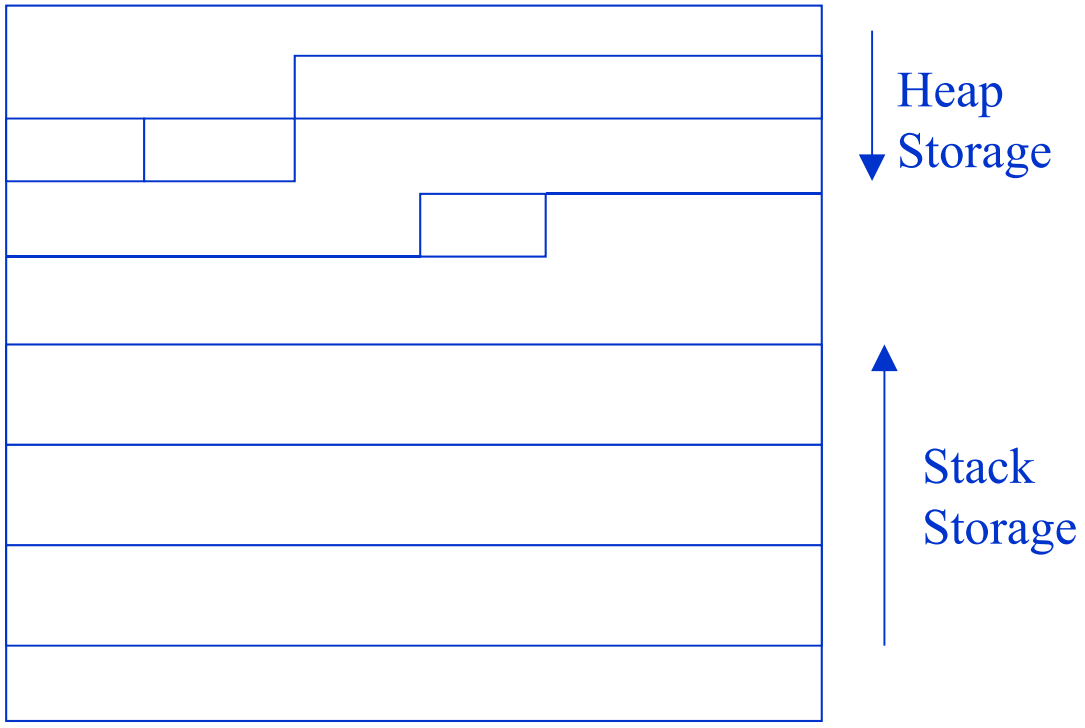
```
int* p;
int* q;
p=new int(5);//allocation and initialization,*p=5
q=new int[10]; //gets q[0] to q[9] with q=&q[0]
delete p;
delete []q;
```

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;
```



Stack and Heap Space



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

Exception Handling

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

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;
}
```

h{fhswlrqWkurz1fss

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？