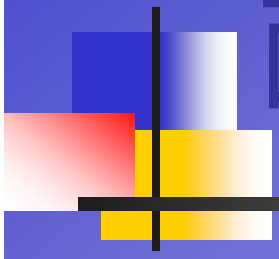
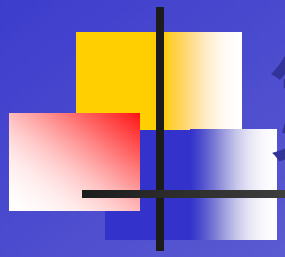


面向对象的程序设计

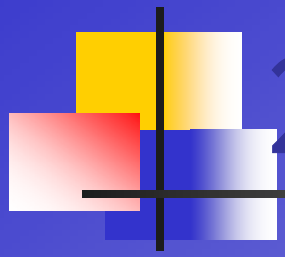
华中科技大学计算机学院
李瑞轩



第2章 C++的变量、类型及函数

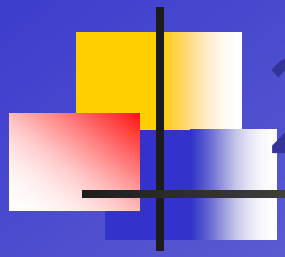
本章内容：

- 2.1 声明及定义
- 2.2 类型定义
- 2.3 引用类型
- 2.4 函数参数
- 2.5 函数内联



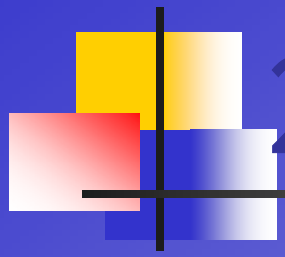
2.1 声明及定义

- 声明：对名及其内涵的不完整描述。
- 定义：对该名及其内涵的完整描述。
- 名可以声明多次，但只能定义一次。
- 类型声明：前向引用声明只说明类名。
  - class Student;
- 变量声明：类型、变量名。定义再加初值。
- 函数声明：函数原型只说明函数名，返回类型以及函数参数。定义再加函数体。



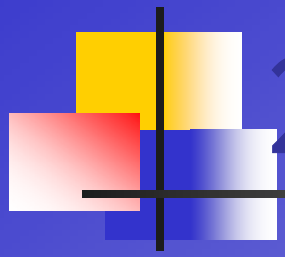
2.1 声明及定义

- C语言：一般只允许以常量表达式初始化变量，局部非静态变量可以用任意表达式初始化。变量声明和定义必在语句前。
- C++语言：所有变量都可以用任意表达式初始化。变量声明和定义不必在语句前。兼容C,局部非静态变量如不初试化，则其值不确定。
- 常量表达式：编译时可计算出常量值。
- 任意表达式：常量、变量、函数调用等组成。只能在运行时计算出值。



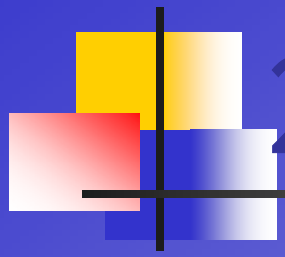
2.1 声明及定义

- 常量表达式：3， 3.14*3*3, size 2, sizeof(int[2]), sizeof(3+printf(“ABC”))
- 任意表达式：
  - 常量表达式
  - **A, 3+A**
  - **printf(“ABCDEF”)**
  - **3+ printf(“ABCDEF”)**
- 注意：int[2]是类型表达式，而非值表达式。值表达式包括常量表达式和任意表达式。值表达式又可分为左值或右值表达式。



2.1 声明及定义

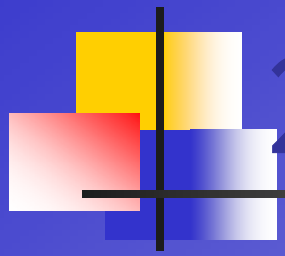
- 左值表达式：可以出现在等号左边的表达式。
  - 非只读类型的变量：int x; x=2;
  - 引用非只读类型的变量：int &y=x; y=3;
  - 指向非只读单元的内容访问：int *p=&x; *p=1;
  - 引用非只读类型的函数：int &f( ){ }; f()=3;
  - 前置++和—运算，赋值运算：(x=2)=3; (++x)=5;
- 右值表达式：只能出现在等号右边的表达式。
- 注意：左值表达式是右值表达式，反之不成立。
- 某些变量或参数传递只能用左值表达式。如非只读类型的引用变量或参数。



2.1 声明及定义

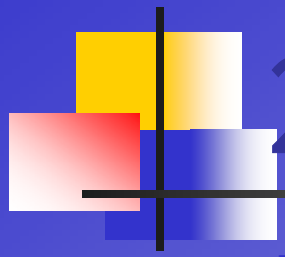
- 【例2.1】C++的声明、定义及初试化。

```
#include <stdio.h>
extern  int  h=0; //C变量定义
extern  int  i;   //C变量声明
int   i;          //C变量定义，缺省初始化i=0
int   j=i+4;      //C++变量定义，初始化j=i+4
static int  p=j+5;//C++变量定义，初始化为p=j+5
```



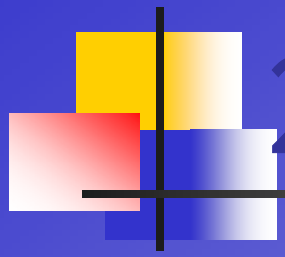
2.1 声明及定义

```
void  main(void){
    static int  n=j+5;          //C++定义，初始化为n=j+5
    int  i=20 ;                 //C定义，初始化为i=20
    int  k;                     //C声明，k未初始化值不定
    for(int j=i+2;j<9;j++)      //C++定义，初始化为j=i+2
    {  int  m=5;                //C定义，初始化为m=5
         k+=m;                  //警告，k未初始化就被引用
    };
    int  q=23;                  //C++在语句中间定义变量
    scanf("%d",&j);             //输入main外定义的变量j
    struct {int k,m;}b={j+3,5};//C++定义，任意表达式初始化
    int a[4]={scanf("%d",&k),1};//C++定义，任意表达式初始化
}
```



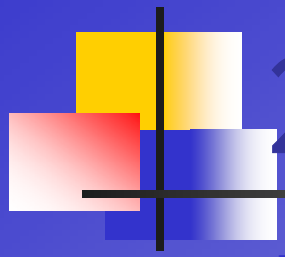
2.1 声明及定义

- C允许初试化局部静态数组。C++的局部自动数组通常在栈段分配空间，若初始化则还会在数据段分配空间。
- 程序X.CPP：
- void main(void) //空间在栈段
- { int array[10]; }
- 程序Y.CPP：
- void main(void) //空间在数据段
- { static int array[10]={1, 2};}
- 程序Z.CPP：
- void main(void) //空间在栈段和数据段
- { int array[10]={1}; }



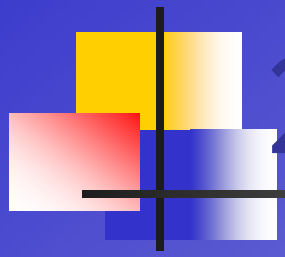
2.1 声明及定义

- 简单类型全局变量：若初始化表达式是任意表达式，则初始值由编译生成的开工函数在运行时计算得到。
- 复杂类型全局变量：即全局对象，其初始化由开工函数调用其相应的构造函数完成。
- 注意：开工函数先于main执行，收工函数后于main执行。收工函数负责自动执行全局对象的析构函数。因此，程序被看作一个对象，有构造（生），活动（main)，析构（死）。



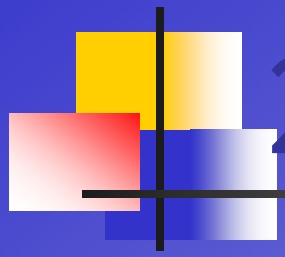
2.1 声明及定义

```
#include <iostream.h>
long sum(void){
    int   m, n, s=0;
    cout<<"Please input:";
    for(cin>>n,m=1; m<=n; m++)   s+=m;
    cout<<"\nSum="<<s;
    return  s;
}
long  x=sum( );//开工函数初始化x调用sum
void main(void){ } //空函数有运行结果
```



2.2 类型定义

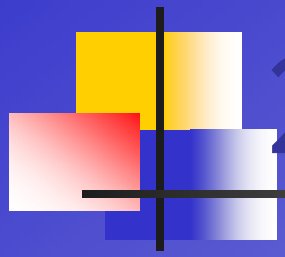
- 根据优先级和结合性解释类型 ：
  - 先解释优先级较高的运算符
  - 优先级相同时则按结合性顺序解释
- 有关优先级的运算符：
  - 星号*、括号( )、函数( )、数组下标[ ]
  - 以后要介绍的成员指针运算符.*和->*
- 注意：
  - ()[]，[][]的结合性自左至右
  - **，*&的结合性自左至右



2.2 类型定义

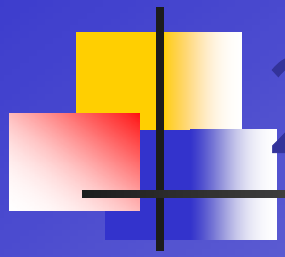
char (f)(int);

- 两个( )，第一个为括号，第二个为函数；
- 优先级相同，根据结合性自左至右解释；
- 先解释左边的(f)，(f)定义了标识符f；
- 再解释(int)说明f是一个有int参数的函数；
- 最后解释char，说明函数f的返回字符型；
- 上述定义等价于char f(int)。(f)没有实质性的作用。



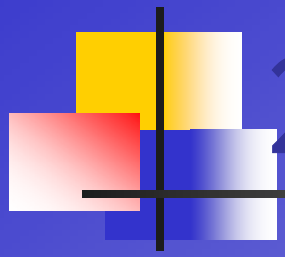
2.2 类型定义

- char (*f)(int);
- (*f)和(int)的优先级相同，两个( )的结合性为自左向右，先解释(*f)再解释(int)。
- ① f是一个指针；
- ② 该指针指向一个函数；
- ③ 该函数有一个int类型的参数；
- ④ 该函数返回一个char类型的值。



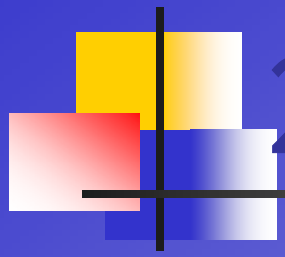
2.2 类型定义

- char *f(int);
- 其中，*的优先级低于(int)，先解释f(int)再解释*。
- ① f是一个函数；
- ② 该函数有一个int类型的参数；
- ③ 该函数返回一个指针类型的值；
- ④ 该指针指向一个字符char。



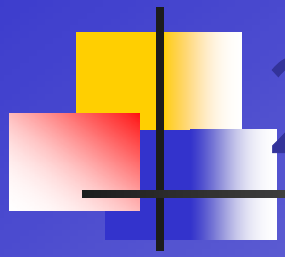
2.2 类型定义

- 括号可提高被解释对象的优先级，根据实际情况适当时可省略或不予解释。
- C和C++规定，函数不能返回一个数组，数组的元素不能是函数。数组的元素可以是指针类型，指针可以指向函数：
  - int (*(**f)[10])(int, int);
- 其中，**自右向左解释，先*f，再**f。 (**f)中( )提高优先级，否则**f[10]将先解释f[10]。



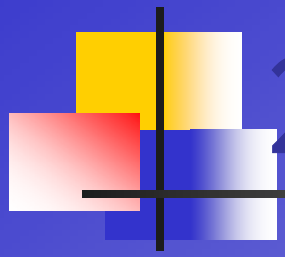
2.2 类型定义

- 根据运算符的优先级和结合性解释typedef定义的复杂类型：
  - typedef int (*F[10])(int, int);
  - F *f;
- 由于F *f表示f指向F，然后再解释F，故将*f代换第一行的F时，应将*f括起来提高优先级，得到等价于如下形式的变量定义：
  - int (*(*f)[10])(int, int);
- 强制类型转换中的类型表达式也必须通过优先级和结合性解释：(int (*(*)[10])(int, int)) x 。



2.2 类型定义

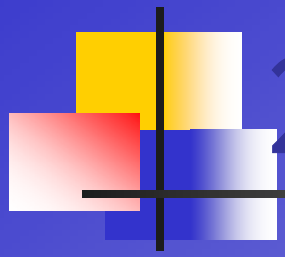
- 类型void是简单类型，表示函数无参或无返回值，返回void的函数是过程，无须返回值：void f(int x)以及void main(void) 。
- 类型void *表示所指对象类型不定，通过void *向对象赋值时必须进行强制类型转换。
- 任何地址常量、变量地址及指针变量的值都可以直接赋给void *类型：delete



2.2 类型定义

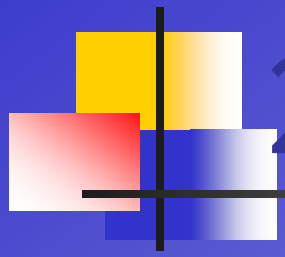
【例2.5】void *类型的指针变量。

```
void  main(void){
    int  p;   float q, *m=&q; void  *r=&p;
    *r=345;            //错误:往类型不定的单元赋值
    *(int *)r=345;   //强制类型转换，p=345
    *(int *)r=3.14;  //强制类型转换，p=3
    r=m;               //任意指针可直接赋给void *指针
    *r=3.14;           //错误:往类型不定的单元赋值
    *(float *)r=3;    //强制类型转换，q=3.00
    *(float *)r=3.14;//强制类型转换，q=3.14
}
```



2.2 类型定义

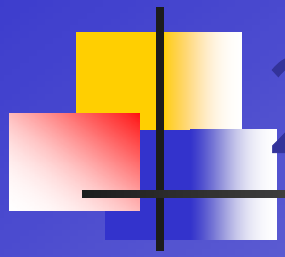
- 保留字const说明只读变量或不变变量,其值程序自身不能修改，故必须有一个初始值，在定义时必须初始化。
- C++提倡用只读变量代替#define定义常量，可避免不当宏替换导致的语法错误。
- 其它程序可能改变只读变量的值。
- 指针变量和指针所指的对象，两者或其中之一都可以定义为只读的 。
- 指针为简单类型



2.2 类型定义

【例2.7】字符串常量指针和字符串变量指针。

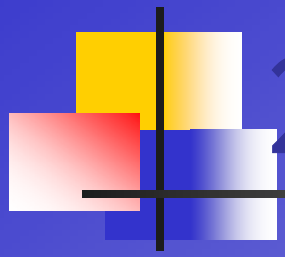
```
void  main(void){
   const char *ptc;                          //非只读变量可不初始化
   char * const cp="Const pointer";//只读变量，必须初始化
   const char *const cptc="Cstptc";//为只读变量必须初始化
   ptc="Pointer to constant";     //非只读指针可修改
   *ptc='P';                                 //出错，*ptc为const char
   cp="Const ptr";                           //出错，cp为const类型
   *cp=“Array of char”[0];          //*cp为char类型，即
  *cp='A'
   cptc="Const to const ptr";     //出错，cptc为const类型
   *cptc='C';                                //出错，*cptr为const char
}
```



2.2 类型定义

指向只读对象的指针可用如下表达式赋值：

- ①指向只读对象的指针变量；
- ②指向普通对象的指针变量；
- ③只读对象的地址；
- ④普通对象的地址；
- ⑤返回只读对象地址的函数调用；
- ⑥返回普通对象地址的函数调用。



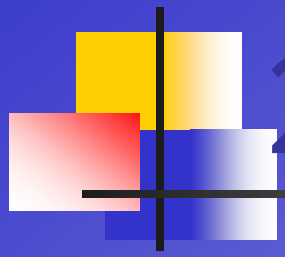
2.2 类型定义

指向普通对象的指针可用如下表达式赋值：

- ①指向普通对象的指针变量；
- ②普通对象的地址；
- ③返回普通对象地址的函数调用。

指向普通对象的指针不能用如下表达式赋值：

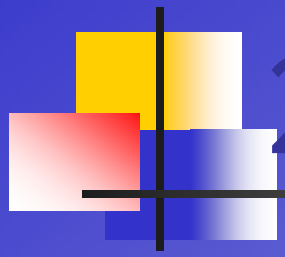
- ①指向只读对象的指针变量；
- ②只读对象的地址；
- ③返回只读对象地址的函数调用。



2.2 类型定义

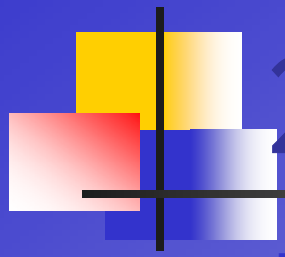
【例2.8】只读对象地址不能赋给普通指针。

```
void  main(void){
    int    i, *p=&i, *h( );
    const  int  j=8, *q=&i, *f( );
    q=&j;          q=p;          q=h( );
    q=f( );        p=h( );
    p=&j;   //错：允许p=&j，则*p=5可修改const j
    p=q;    //错误，由上类推
    p=f( );//错误，由上类推
}
```



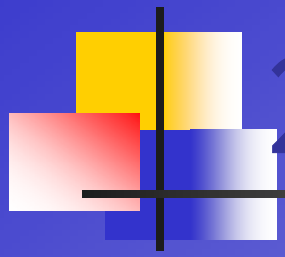
2.2 类型定义

- **枚举类型**：由enum定义，通常以int实现。
- **枚举元素**：实现为只读整型变量或整型常量。可以指定元素值，整型值可以重复，第一个缺省值=0，下一个在上一个的基础上递增1。
  - **enum WEEKDAY {Sun, Mon, Tue, Wed, Thu, Fri, Sat};**
- 上述枚举定义等价于：
  - **typedef int WEEKDAY;**
  - **const int Sun=0, Mon=1, Tue=2；**
  - **const int Wed=3, Thu=4, Fri=5, Sat=6;**
- 可以定义如下枚举及变量：
  - **WEEKDAY w1=Sun，w2(Mon);//等价于w2=Mon**
  - **enum WK {Su=-2, Mo, Tu, We, Th, Fr =1, Sa}; //We=1**



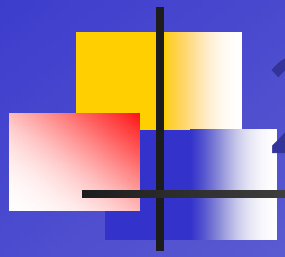
2.2 类型定义

- 挥发变量：由volatile声明和定义。从程序自身的角度来看，挥发变量的值可以自发改变(其实由其他进程或程序改变)。挥发变量可同时为只读的。
  - **const volatile int y=3;**
  - **volatile int x;**
  - **x=3;**
  - **if(x==4) cout<<"x changed by other routines";**
- 【例2.9】编程唱“亚洲雄风”。进程int1c和main并发访问挥发变量number。函数main检测number的值，等到唱完一曲后结束。



2.3 引用类型

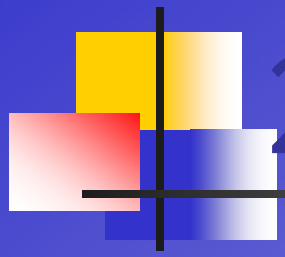
- 引用类型：用运算符&声明和定义，某个实体的别名(不是新实体，逻辑上不分配内存)。变量、函数参数和返回值都可为引用类型。
- 只读引用变量：引用常量或只读变量，必须立即用右值表达式初始化。
- 普通引用变量：引用普通(左值)变量，必须立即用左值表达式初始化
- 挥发引用变量：引用挥发(左值)变量，必须立即用挥发变量的左值表达式初始化。注意C++的版本。



2.3 引用类型

引用逻辑上不分配内存，编译为指向被引用实体的指针。

- swap(int *x, int *y){
- int t=*x;
- *x=*y;
- *y=t;
- } //分开编译后，两个函数代码完全一样
- swap(int&x, int&y){
- int t=x;
- x=y;
- y=t;
- } //引用传递实参的指针，实参为数组时节省空间



2.3 引用类型

- 引用参数：只读引用参数，普通引用参数。不能在函数体内改变只读引用实参的值。
- 只读引用参数：调用时要用值表达式初始化，不能带出函数的运算结果。
- 普通引用参数：又指左值引用参数，相当于换名形参，调用时必须用左值表达式初始化，可以带出函数的运算结果。

2.3 引用类型

【例2.11】将命令行转换为十进制整数打印。

```
#include <iostream.h>
int strint(const char *str, int &val)
{  char  c;                    //str是只读引用吗？不是，可
   val=0;
   while(c=*str++) //改变str的值
   if((c>='0')&&(c<='9')) val=val*10+c-'0';
   else  return 0;
   return 1;
}
```

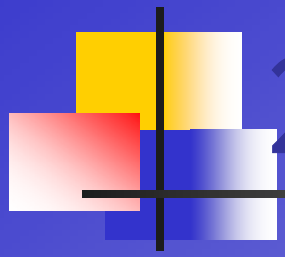
2.3 引用类型

- void main(int argc, char *argv[ ])
- **{ int i, n;**
- **for(i=1; i<argc; i++)**
- **if(strint(argv[i], n))cout<<"Argv["<<i<<"]="<<n<<"\n";**
- **else cout<<"Argument "<<i<<" is not a decimal";**
- **}**
- strint(argv[i], n)返回时左值变量n带回结果。
- 类型转换的结果为右值而非左值。左值引用参数要用同类型的左值表达式实参，否则编译产生同类型的左值临时变量。

2.3 引用类型

- 【例2.12】定义函数实现++的后置运算。

```
int  inc(int &x) {  return  x++; }
int  i=1, j;        char c=1;
void main(void){    //后置运算、加法运算结果为右值，生成
   j=inc(i);              //i=2，j=1，   i带回结果
   j=inc(++i);            //i=4，j=3，   i带回结果
   j=inc(i++);            //i=5，j=4，   匿名变量带回结果而不是i
   j=inc(i+5);            //i=5，j=10，  匿名变量带回结果而不是i
   j=inc(c);              //c=1，j=1，   匿名变量带回结果而不是c
   j=inc(++c);            //c=2，j=2，   匿名变量带回结果而不是c
   j=inc(c++);            //c=3，j=2，   匿名变量带回结果而不是c
   j=inc(c+5);            //c=3，j=8，   匿名变量带回结果而不是c
} //左值c从char转换为右值int，而int &x要左值，生成
```



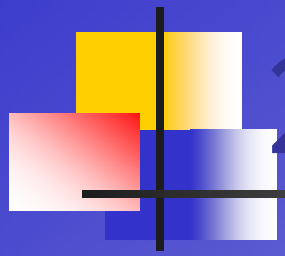
2.3 引用类型

【例2.13】访问引用变量实质上访问被引用实体

```
int  i;                    //全局变量缺省初始化i=0
const int &j=2;//只读引用变量引用常量
int &x=++i;      //++i为左值，被引用变量为i
int &y=i=0;      //i=0为左值，被引用变量为i
int &z=y=3;      //y=3为左值，被引用变量为i
int &f(int &v){ return v+=5;}//返回v引用的变量
void main(void){
    int i=0;
```

2.3 引用类型

```
const int k=10, &j=k;
const int &m=2;//引用值为2的(只读变量)常量
const int &n=i;  //生成临时匿名变量
int &x=i++;      //警告，右值时临时生成匿名变量
int &y=++++i; //++++i为左值，y引用变量i
int &z=i=4;      //i=4为左值，z引用变量i
int &r=z=8;      //z=8为左值，同类型赋值r引用i
i=3;             //r=y=z=i=3，x=0
x=6;             //x=6，r=y=z=i=3
r=12;            //r=y=z=i=12，x=6
++y=10;          //左值++y代表i，10赋给i使r=y=z=i=10
(z=10)=15;       //左值z=10代表i，15赋给i使r=y=z=i=15
(f(r)=1)=2;     //f(r)返回r引用的i，(i=1)为左值r=y=z=i=2
}
```



2.3 引用类型

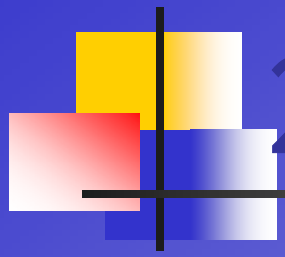
【例2.14】利用函数返回引用改变局部静态变量的值。

```
#include <stdio.h>
char *&weekday(int &day){
   static char *week[7]={"Sun","Man","The","Wed","Thu","Fri","Sat"};
   return week[day++];
}
void main(void){
   int day=0;
   printf(weekday(day));  //打印Sun，执行后day=1
   weekday(day)="Mon"; //将被引用的week[1]指向"Mon“, day=2
   weekday(day)[1]='u';  //将"The"改为"Tue"，执行后day=3
   printf(weekday(day+2));//day+2非左值,生成临时变量,day=3
   printf(weekday(day=0)); //打印Sun，执行后day=1
}
```

2.3 引用类型

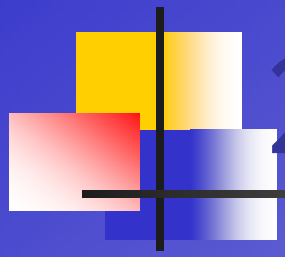
【例2.15】主调函数引用被调函数的自动变量导致程序结果不确定。

```
#include <iostream.h>
int  &f(int i){      //返回引用基于栈的变量i
    int &j=i;        //引用基于栈的自动变量i
    return j; }
int &g( ){           //返回引用基于栈的变量k
    int  k=6, &m=k; //引用基于栈的变量k
    return m; }
void main(void){
    int &x=f(10), x1=x;
    cout<<"x="<<x<<" x1="<<x1<<"\n";    //x==x1? No
    int &y=g( ), y1=y;
    cout<<"y="<<y<<" y1="<<y1<<"\n"; } //y==y1? No
```



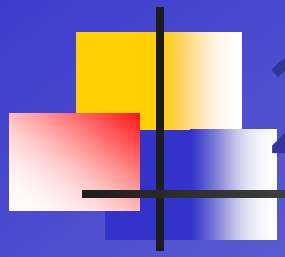
2.3 引用类型

- 引用变量是被引用实体的别名，逻辑上不分配存储单元。访问引用变量访问该实体。
  - 数组元素不能为引用类型，否则数组空间不存在
  - 指针不能指向引用变量，逻辑上引用变量不分配存储单元
- 被引用实体必须是有地址分配存储单元的实体
  - 寄存器变量可以被引用，寄存器变量被编译为自动变量
  - 位段类型的成员分量不能被引用，计算机没有按位编址，而是按字节编址
  - 引用变量逻辑上没有分配单元，不能作为实体引用



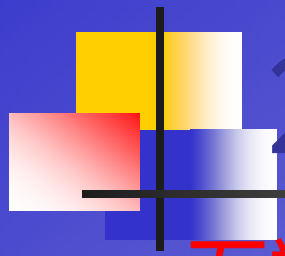
2.3 引用类型

- 【例2.16】引用变量的用法。
- struct A{
- int i:3;
- int j:4;
- int k;
- } a;
- int j(0); //C++的构造初始化形式j=0
- int &k=j++; //错误：须以左值初始化



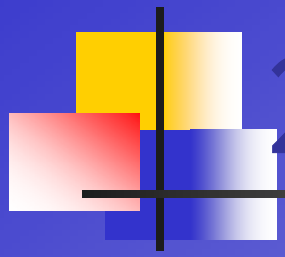
2.3 引用类型

```
void main(void){
    register int i=0, &j=i;//i, j都编译为自动变量
    int  &&m=j;  //错误，不能引用引用变量j
    int  &*m; //错误，不能指向不分配内存的引用
    int  &s[4]; //错误，数组元素不能为引用变量
    int  t[6], (&u)[6]=t;//引用变量u可以引用数组t
    int  &v=t[0]; //引用变量v可以引用数组元素
    int  &w=a.j;  //错误：位段无地址，不能被引用
    int  &x=a.k;  //a.k不是位段
}
```



2.4 函数参数

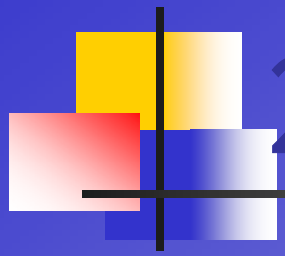
- **函数类别**：普通函数、友元函数、成员函数、构造函数、析构函数，虚函数，纯虚函数
- **普通函数**：C语言的函数
- **函数原型**：关于函数名，返回类型，函数参数的描述。函数参数可以没有名称，但不许说明类型。
- **重载函数**：同名函数在参数个数或类型上有所不同。老版本C++用overload声明重载。析构函数的原型是固定不变的，不能重载。重载与函数的返回类型无关。
- **内联函数**：调用时可以将函数体插入到被调用位置的函数。而不用在被调用位置用call指令。



2.4 函数参数

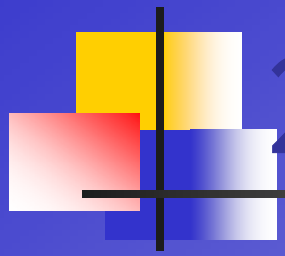
- 【例2.7】打印当前系统时间。

```
#include <dos.h>
#include <iostream.h>
overload GetTime;     //本行可以省略
long GetTime(void){...}
long GetTime(int &hours, int &minutes, int &seconds){...}
void main(void){
    int  h, m, s;
    cout<<"Now is "<<GetTime(h, m, s)；
    cout<<" seconds from midnight,";
    cout<<"Or "<<h<<":"<<m<<":"<<s<<"\n";
}
```



2.4 函数参数

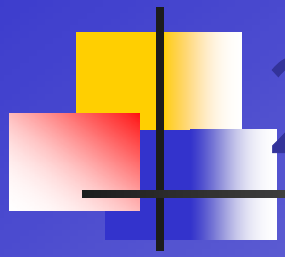
- 省略参数用省略号…表示，即可以有任意个任意类型的参数: int printf(const char *format, …);
- long sum(int n, ...){
- long s=0;
- int *p=&n+1;
- for (int k=0; k<n; k++) s+=p[k];
- return s;
- }
- void main( ){ int a=4; long s=sum(3,a,2,3); }



2.4 函数参数

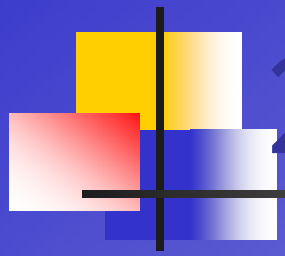
【例2.20】缺省时在屏幕指定坐标打印信息。

```
#include <conio.h>
#include <stdio.h>
int sayatxy(char *msg, int x=1, int y=1){
    gotoxy(x, y);                    //将光标移到x列，y行
    return  printf(msg);             //打印msg
}
void main(void){
    sayatxy("Default");              //在(1,1)处打印Default
    sayatxy("Default Line", 30);//在(30,1)处打印Default Line
    sayatxy("Specified",1,2);     //在(1,2)处打印Specified
}
```



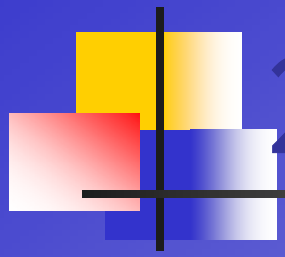
2.4 函数参数

- 缺省参数：在没有传递实参时使用缺省值的参数。
- 使用方法：可以用任意类型的表达式指定缺省值，但表达式中不能出现同一个参数表的参数，所有缺省参数必须出现在非缺省参数的右部。不能同时在函数的原型声明和函数定义中定义缺省参数。



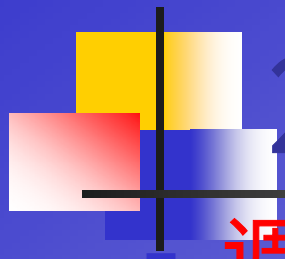
2.4 函数参数

- int u, v, m(int, int);
- int a(int x=5,int y=m(u,v));//用调用指定缺省值
- int b(int x=1,char,int z=1);//错：x在非缺省参数左边
- int a(int,int y=1,int); //错：缺省参数y不在最右部
- int b(int x=3); //声明b(int)缺省值x=3
- int b(int x=3) {return x*x;}//错：不能再次定义x=3
- int f(int x, int y=x++);//错：表达式有同参数表的参数
- 注意：调用f(x)的实际调用形式为f(x, x++)。对于x=3，自左至右计算函数参数的调用为f(3, 3)，自右至左计算函数参数的调用为f(4, 3)，这种不一致会导致程序的不可移植性。



2.4 函数参数

- **参数匹配**：通过匹配实参和形参找到参数个数及类型完全一致的唯一函数调用。若匹配结束找不到这样的函数，或者找到多个匹配函数，编译程序都将报错。
- 【例2.21】error(“System A”)可匹配：
  - **error(char *sys, char*msg="Undefined error\n")**
  - **error(char *sys, int code=0)**
  - **error(char *sys, …)**
  - **error(char *sys)**



2.5 函数内联

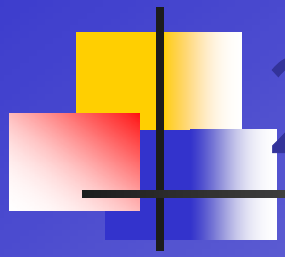
- 调用开销：调用(压栈传递实参 ，调用指令，压栈保护寄存器)和返回(出栈恢复寄存器，返回指令，出栈丢弃实参)的开销。函数体越小，相对开销越大。调用开销10指令，程序有100个调用位置：
  - 函数体5指令，程序长：100*10+5=1005
  - 函数体20指令，共执行100*10+20=1020
- 函数内联：用保留字inline声明或定义，调用时在调用处直接插入函数体。一般对函数体较小的函数内联，太大则反而使编译后程序加长：
  - 函数体5指令，程序长100*5=500
  - 函数体20指令，程序长100*20=3000

2.5 函数内联

【例2.22】编程计算圆的面积。

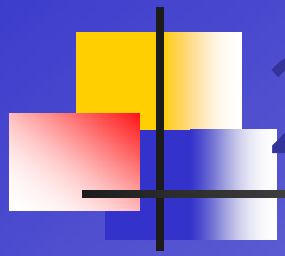
```
#include <iostream.h>
inline double area(double r){
    return  3.1415926*r*r;
}
void main(void){
    double   m;
    cout<<"Please input a radius:";
    cin>>m;
    cout<<"The area is "<<area(m);
}
```

将直接计算3.1415926*m*m
而不用call area指令调用函数



2.5 函数内联

- 内联失败：不能在调用处插入函数体。原因：
  - 包含产生分支转移的语句：if, switch,while, for,do while,? :, 函数调用
  - 内联函数的定义出现在调用之后
  - 有关于取内联函数地址的指令
  - 内联函数同时被定义为虚函数或纯虚函数
- 注意事项：
  - 内联使函数的作用于局部于当前程序文件，相当于在函数前使用了static。
  - 全局main函数不能内联，局部main函数可以内联。



2.5 函数内联

- 版本相关：本例同编译程序的版本相关。假定一程序有两个程序文件构成，A.CPP如下：
  - **extern int f( );**
  - **int main( ){ return f( ); }**
- 程序文件B.CPP如下：
  - **inline static int main( ) { return 3; } //static可省略**
  - **int f( ){ return main( ); }**
- 编译连接A.cpp和B.cpp生成M.exe，入口main函数在A.cpp中。