

Variables and Operators in classes

We have to know how to:

- Declare**
- Use the built-in C++ data types**
- Create user-defined types**
- Add member variables to a class**
- Assign values to a variable**
- Cast (change) the type of a variable**

The fundamental data types

Bool	in .NET only	true	false
Char		-128	127
Short		-32768	32767
Int		2^{32}	
Long	like int – twice for some compilers		
_int8	.NET – same as char		
_int16	.NET – same as short		
_int32	.NET – same as int		
_int64	.NET – 64 bits long		
Float	for example 3.7	in Visual C++ up to	6 decimal places
Double		up to 15 decimal places	
Poiners		ex: int*	
Array	example: int[]		
Reference type	example: double&		

Declaring variables

- Before using them

- Examples:

```
int myFirst;  
double x, y, z;  
long Salary = 0;  
unsigned int I;
```

// qualifier limits the variable to positive numbers

- Declaring results to:
 - Allocate memory
 - Reserve the name
 - Link the name with the declared type

Enumerations

- for variable that can take only specific set of values
- keyword enum

```
enum WorkDays {Monday, Tuesday, Wenesday, Thursday, Friday}  
const int Saturday = 7;  
WorkDays workday;  
workday = Thusday;           //OK  
workday = Saturday;         //wrong!
```

TypeDef

- user-defined synonym for an existing type

```
typedef unsigned int positivenumber;
```

```
....
```

```
positivenumber one, two;
```

Operators and Expression

- all the expressions give a result

`x = salary + bonus;`

- assignment operators

`animals = dogs = cats = 0;`

- arithmetic operators

`+ - * / % += -= *= /=`

`%=`

`a = a + 5;` is equal to `a +=5;`

`++`

`--` postfix or prefix

Operators and Expression

- relational and logical operators

< >= <= == !=

returns true or false

&&

\|

!

- bitwise operators

& \ ^ (exclusive OR) ~ >>

<<

for char, short, int , long values

```
int a = 5; // 101
```

```
a = a << 2; // 10100 that is 20
```

Assignment

- Values are stored into fields (and other variables) via assignment statements:
 - *variable = expression;*
- A variable stores a single value, so any previous value is lost.

ternary operator ?:

```
int a; bool b = true;  
a = b ? 1 : 2; // b is true, so a = 1  
b = false;  
a = b ? 1 : 2; // b is false, so a = 2
```

Type casting

- old C style casting: (float) 7;
- `static_cast<type>(variable);` //for normal change of type
- `const_cast<type>(variable);` // change the type of const variables
- `dynamic_cast<type>(variable);` // cast object down or across the inheritance hierarchy
- `reinterpret_cast<type>(variable);` //convert the pointer's type

example: `int a = 10; double b;`
 `b = (double) a; b = static_cast<double>(a);`

Pointers

```
int* pi;  
double* pd;  
char* pc;
```

```
int x = 10, y=0;  
int* pX = NULL;           //declare a pointer  
pX = &x;                  // take the address of x  
y = *pX;  
// dereferencing – now y has the value of x
```

```
*pX = 20;  
// dereference operator – x is now 20
```

Псевдоними

- Псевдонимът представлява неявен указател, който играе ролята на друго име за дадена променлива.
- Съществуват 3 начина, по които може да се използва даден псевдоним:
 - Псевдонимът може да бъде предаван на функция
 - Псевдонимът може да бъде връщан като резултат от функция
 - Може да бъде създаван независим псевдоним

Псевдоним като параметър на функция

- Когато един обект се предава като параметър на функция се създава копие на този обект.
- Конструкторът на параметъра не се извиква, но се извиква неговият деструктор, когато функцията върне резултат.
- Едно от решенията за този проблем е обекти да се предават чрез **псевдоними**. Когато обекта се подава чрез псевдоним, не се създава копие на обекта, а следователно не се извиква и деструкторът, когато функцията върне резултат.

Псевдоним като параметър на функция

- Промените, които се направят с обекта в рамките на функцията се запазват и след излизането от функцията.
- Ако обектите не трябва да бъдат променени от функцията, то се използват **константни псевдоними**.

```
double dist (const point &p1, const point &p2);  
//Разстояние между две точки
```

Пример 1/3

```
class point                //Клас точка
{ private:
    double x;
    double y;
public:
    point(double xcoord, double ycoord);
//Конструктор с два параметъра
    double dist (const point &p1, const
point &p2);
//Разстояние между две точки
    ~point();
//Деструктор
};
```

Пример 2/3

```
point::point(double xcoord, double ycoord)
{
    x = xcoord;
    y = ycoord;
}
double point::dist(const point &p1, const point
&p2)
{
    return sqrt((p1.x-p2.x) * (p1.x-p2.x) +
                (p1.y-p2.y) * (p1.y-
                p2.y));
}
point::~~point() //Деструктор
{ cout << "Destructing ";
  cout << endl;
}
```

Пример 3/3

```
void main()  
{  
    point a(3,4), b(10,4);  
    cout << a.dist(a,b) <<  
endl;  
}
```


Псевдоним на обект като върнат резултат от функция

- Функция може да върне псевдоним на обект като резултат.
- Функция, която връща псевдоним като резултат, може да се използва и от лявата страна на оператора за присвояване.

Пример: масив с проверка на границите 1/3

```
class Array {  
    int size;  
    char *p;  
public:  
    Array(int num);  
    ~Array() { delete [] p; }  
    char &put(int i);  
    char get(int i);  
};
```

Пример: масив с проверка на границите 2/3

```
Array::Array(int num)
{
    p = new char [num];
    if(!p){
        cout << "Allocation error\n";
        exit(1);
    }
    size = num;
}
char &Array::put(int i)          //Поставя се нещо в
    масива
{
    if(i<0 || i>=size) {
        cout << "Bounds error!!!\n";
        exit(1);
    }
    return p[i];                //връща псевдоним за p[i]
}
```

Пример: масив с проверка на границите 3/3

```
char Array::get(int i)          //Взема се нещо от масива.
{
    if(i<0 || i>=size) {
        cout << "Bounds error!!!\n";
        exit(1);
    }
    return p[i];                //връща символ
}
int main()
{
    Array a(10); a.put(3) = 'X'; a.put(2) = 'R';
    cout << a.get(3) << "\n" << a.get(2);
    cout << "\n";
    a.put(11) = '!'; //грешка по време на изпълнение
    return 0;
}
```

References



In different situation transferring the complex object (like string or...) is ineffective. When parameter substitution is the case, or returning a string a copy of the object is effectively used. The original is unchanged. A big amount of data is transferred. So, some kind of pointer must be used. This is named "reference". The reference become another object name. (if T is the object, the reference is T&)

- *a reference variable is an alias for another variable*
- *all reference variables **must be initialized** when declared*

- ```
int x = 10;
int& y = x; // declare y as reference to x
y = 4; // change the value of x to 4
(now &x == &y)
```

- *reference variables are very important when passing variables to functions*
- *once defined as T&, the reference must be initialized with a "T" object or with an object, that can be transformed to "T"*
- *after initialization , reference can't be reinitialized with another content.*

# Константи

- Към дефиницията на всяка променлива може да се прилага модификатора ***const***.
- Указва, че обекта не може да се променя.
- Води до грешка при компилация, в случай че се опитаме да променим ***const*** обект.
- Константи задължително трябва да се инициализират.
- Пример:  

```
const Point origin (0.0 ,0.0);
origin.set_x (2.0); // грешка!
origin.get_x (); // грешка???
```

# Указатели и константи

- При операциите с указатели участват два обекта – самият указател и обекта, към който сочи указателя.
- Когато ключовата дума **const** се постави пред дефиницията на указателя, това означава че константен е обекта към който сочи указателя.
- За да се декларира, че самият указател е константен, се използва **\*const**, вместо **\***.

# Пример

```
char str1 []= " hello " ;
char str2 []= " hell " ;
const char * pc= str1 ;
pc [2]= 'a' ;
pc= str2 ;
char *const cp= str1 ;
cp [2]= 'a' ;
cp= str2 ;
const char *const cpc = str1 ;
cpc [2]= 'a' ;
cpc = str2 ;
```



# Константни член-функции

- За да може една член-функция да се прилага към константен обект, компилатора трябва да е информиран, че тази член-функция не променя състоянието на обекта.
- За тази цел се използват **const** член-функции.
- Когато една член-функция е **const**, то в нейната дефиниция не може да се променя състоянието на обекта.
- За дефиниране на една член-функция като константна се използва ключовата дума **const**:
  - В прототипа на функция след списъка от параметри.
  - В дефиницията на функцията преди тялото на функцията.

# Пример

```
class Point {
 double x_, y_;
 public:
 ...
 double get_x () const {
 return x_;
 }
 double get_y () const ;
 ...
};

double Point :: get_y () const {
 return y_;
}
```

# Константни член-функции

- Когато една член-функция е дефинирана като константна, тя не може да променя състоянието на обекта, за който е извикана.

```
class Point {
 double x_, y_;
 public:
 ...
 void set_x (double x) const {
 x_=x; // грешка!!
 }
 ...
};
```

# Пример

```
class Point {
 double x_, y_;
 public:
 Point (double x=0.0 , double y =0.0)
 : x_(x), y_(y)
 {}
 double get_x (void) const { return x_;}
 double get_y (void) const { return y_;}
 void set_x (double x) {x_=x;}
 void set_y (double y) {y_=y;}
};
```