Basic C++

Pointers and References

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

Agenda

- Arrays
- Pointers
- Pointer arithmetic
- References
- Constants
- Functions and parameter passing
- Operators
- Function/operator overloading

Arrays

- An array is a strictly continuous memory area.
- Arrays do not know their size, but: sizeof(t) / sizeof(t[0])
- Array names could be converted to pointer value to the first element.
- No multidimensional arrays. But there are arrays of arrays.
- No operations on arrays, only on array elements.


```
int t[2][4];

assert(sizeof(t) == 8*sizeof(int));
assert(sizeof(t[0] == 4*sizeof(int));
assert(sizeof(t[0][0])==sizeof(int));

t[0][1] = t[1][1];
// t[0] = t[1];  syntax error
```

Array

- Array parameters are passed as pointers to the first element
- Array parameters are automatically converted to pointer decl.
- Only the leftmost dimension can be skipped at declarations

```
void g(int t[2][3]);  // 2 is ignored
void g(int  t[][3]);  // same as above
void g(int (*t)[3]);  // same as above, this is the reality

void f()
{
    int t[2][3] = { {0,1,2}, {3,4,5} };  // 6 elements

    g(t);  // pointer to {0,1,2} with type int[3] is passed
}
```

Array

- C++ arrays does not have length property
- Using sizeof() operator we can measure the size

```
void f()
{
    int t[] = { 0,1,2,3,4,5 };  // 6 elements

    for ( int i = 0; i < sizeof(t)/sizeof(t[0]); ++i)  // # elements
    {
            std::cout << t[i] << ' ';
    }
}

void g(int t[])  // in reality: g(int *t)
{
    for ( int i = 0; i < sizeof(t)/sizeof(t[0]); ++i)  // mistake!
    {
            std::cout << t[i] << ' ';
    }
}
```

Array decay

- Array to pointer conversion (the first of the “standard conversions”)
- Reference binding can avoid decay

```
void  aDecay( int *ap)     {std::cout << sizeof(ap) << "\n";}
void  pDecay( int (*p)[6]) {std::cout << sizeof(p)  << "\n";}
void noDecay( int (&a)[6]) {std::cout << sizeof(a)  << "\n";}
template <typename T, int N>
void tNoDecay(T (&a)[N]) { std::cout << sizeof(T) << " " << N
                                     << " " <<sizeof(a) << "\n"; }

void f()
{
  int t[6] = { 0,1,2,3,4,5 };
  std::cout << sizeof(t) << "\n";  // 24

  aDecay(t);                       //  8
  pDecay(&t);                      //  8
  noDecay(t);                      // 24
  tNoDecay(t);                     //  4 6 24
}
```

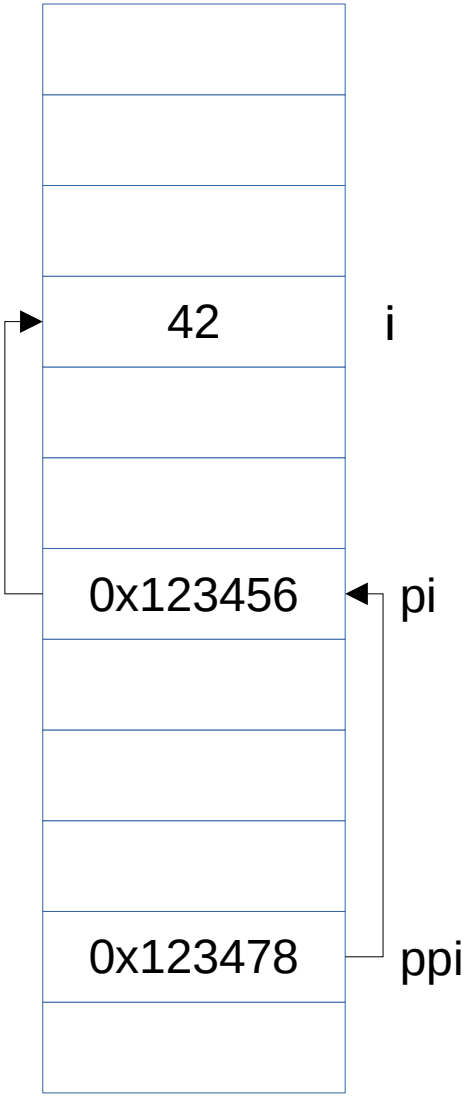
std::array

```
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>

int main()
{
    // construction uses aggregate initialization
    std::array<int, 3> a1{ {1, 2, 3} }; // double-braces required in C++11 (not in C++14)
    std::array<int, 3> a2 = {1, 2, 3};  // never required after =
    std::array<std::string, 2> a3 = { std::string("a"), "b" };

    // container operations are supported
    std::sort(a1.begin(), a1.end());
    std::reverse_copy(a2.begin(), a2.end(), std::ostream_iterator<int>(std::cout, " "));

    std::cout << '\n';

    // ranged for loop is supported
    for(const auto& s: a3)
        std::cout << s << ' ';

    std::array  a1{"foo"};          // C++17 CTAD: std::array<const char*,1>{"foo"}
    auto a2 = std::to_array{"foo"}; // std::array<char,4>{'f','o','o','\0'};
}
```

Pointers

- Pointer is a value that refers to a(n abstract) memory location
- Pointers can refer to ANY valid memory locations

```
int     i = 42;
int   *pi = &i;
int **ppi = &pi;

    *pi = 43; // i == 43
  ++*pi;      // i == 44
++**ppi;      // i == 45
```



Pointers

- **nullptr** (NULL) is a universal null-pointer value
- Non-null pointers are considered as TRUE value

```
char *ptr = getchar("PATH");
if ( NULL != ptr )
{
   // *ptr can be used here
}
```

Pointers

- **nullptr** (NULL) is a universal null-pointer value
- Non-null pointers are considered as TRUE value

```
char *ptr = getchar("PATH");
if ( NULL != ptr )
{
   // *ptr can be used here
}
```

```
char *ptr = getchar("PATH");
if ( ptr )
{
   // *ptr can be used here
}
```

Pointers

- **nullptr** (NULL) is a universal null-pointer value
- Non-null pointers are considered as TRUE value

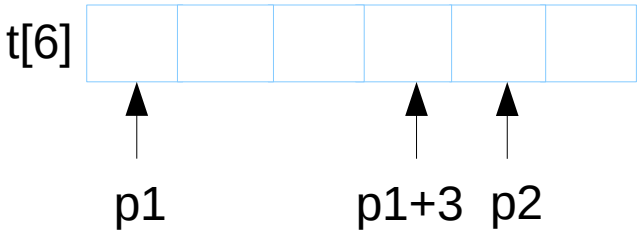
```
char *ptr = getchar("PATH");
if ( NULL != ptr )
{
   // *ptr can be used here
}
```

```
char *ptr = getchar("PATH");
if ( ptr )
{
   // *ptr can be used here
}
```

```
if (char *ptr = getchar("PATH"))
{
   // *ptr can be used here
}
```

Pointer arithmetics

- Integers can be added and subtracted from pointers
- Pointers pointing to the same array can be subtracted
- Pointer arithmetics depends on the pointed type!
- No pointer arithmetics on void *




```
int t[6];
int *p1 = &t[0];
int *p2 = &t[4];

assert( &t[3] == p1+3 );
assert( p2 - p1 == 4 );
assert( p1 + 4 == p2 );

  p = &t[0];
  p = t;
  p + k == &t[k]
*(p + k) == t[k] // *(t+k)
```

Pointers

- Pointers can be used like arrays
- Defined by pointer arithmetics

```
int t[10];

int *ptr = &t[0];

ptr[0] = 42;   // *ptr     = 42;
ptr[1] = 43;   // *(ptr+1) = 43;
ptr[2] = 44;   // *(ptr+2) = 44;
// ...
ptr[9] = 51;   // *(ptr+9) = 51;

ptr[10] = 52;  // ptr[10] is invalid!
```

Pointers and arrays

- Pointers and array names can be used similarly in expressions

```
int t[10];
int i = 3;

int *p = t;   // t is used as pointer to the first element
```

```
*(p+i) == p[i] == t[i] == *(t+i)
```

But pointers ARE NOT EQUIVALENT TO arrays !!!

Pointers and arrays

```
// s1.c
#include <stdio.h>

int t[] = {1,2,3};

void f( int *par)
{
    std::cout << par[1];
    std::cout << t[1];
}
int main()
{
    int *p = t;
    std::cout << t[1];
    std::cout << p[1];

    f(t);
    g(t);
}
```

```
// s2.c
#include <stdio.h>

extern int *t;

void g( int *par)
{
    std::cout << par[1];
    std::cout << t[1];
}
```

Pointers and arrays

```
// s1.c
#include <stdio.h>

int t[] = {1,2,3};

void f( int *par)
{
    std::cout << par[1];
    std::cout << t[1];
}
int main()
{
    int *p = t;
    std::cout << t[1];
    std::cout << p[1];

    f(t);
    g(t);
}
```

```
// s2.c
#include <stdio.h>

extern int *t;

void g( int *par)
{
    std::cout << par[1];
    std::cout << t[1];  // CRASH!!!
}
```

Pointers and arrays

```
// s1.c
#include <stdio.h>

int t[] = {1,2,3};

void f( int *par)
{
    std::cout << par[1];
    std::cout << t[1];
}
int main()
{
    int *p = t;
    std::cout << t[1];
    std::cout << p[1];

    f(t);
    g(t);
}
```

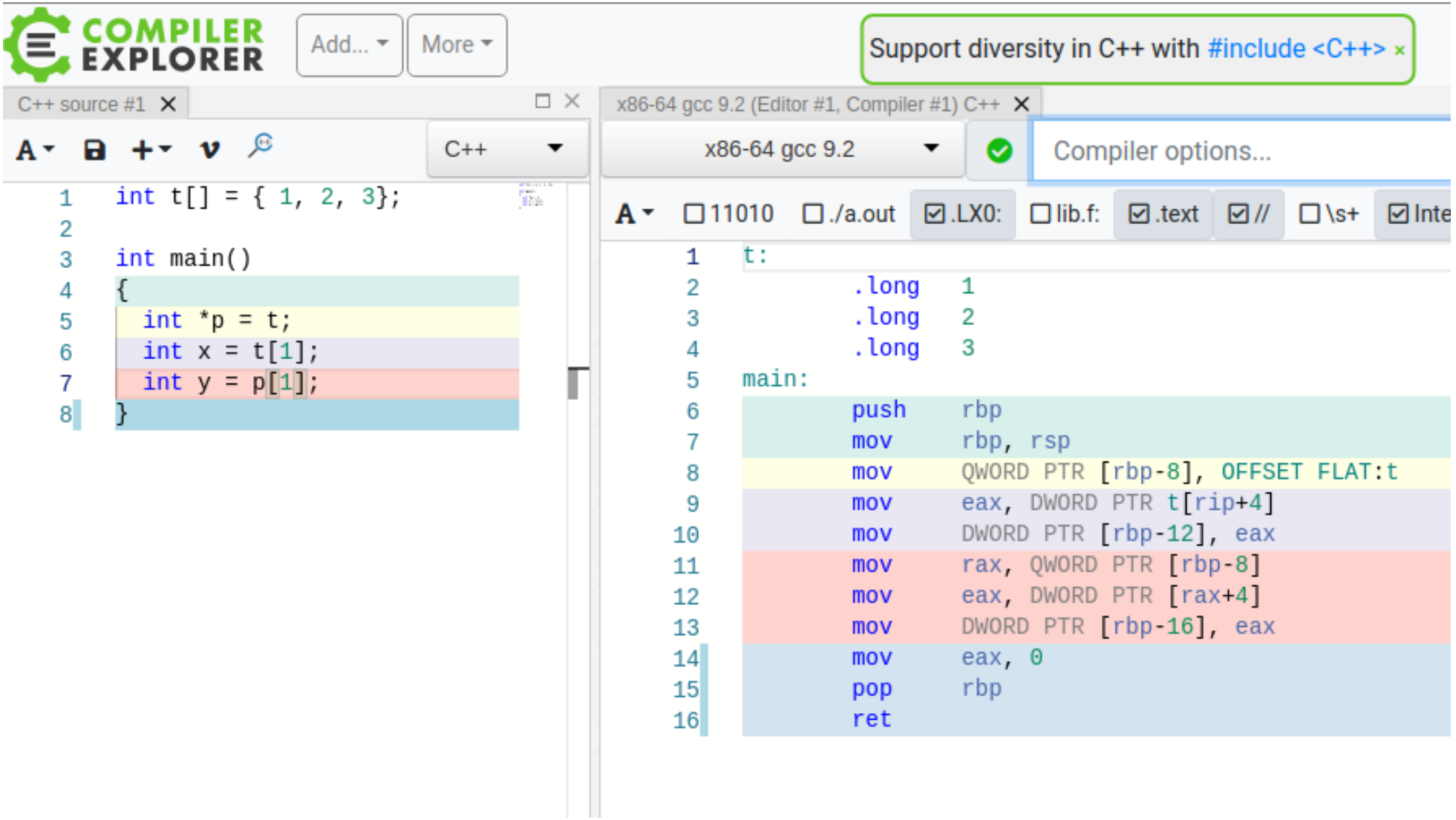
```
// s2.c
#include <stdio.h>

extern int *t;

void g( int *par)
{
    std::cout << par[1];
    std::cout << t[1];
}
```

Pointers and arrays

```
// s1.c
#include <stdio.h>

int t[] = {1,2,3};

void f( int *par)
{
    std::cout << par[1];
    std::cout << t[1];
}
int main()
{
    int *p = t;
    std::cout << t[1];
    std::cout << p[1];

    f(t);
    g(t);
}
```

```
// s2.c
#include <stdio.h>

extern int t[];

void g( int *par)
{
    std::cout << par[1];
    std::cout << t[1];  // WORKS
}
```

Pointers and arrays



Nullptr

- nullptr is a new literal since C++11 of type std::nullptr_t
- Helps to overload between pointers and integer
- Automatic conversion from null pointer of any type and from NULL

```
void f(int*); // 1
void f(int);  // 2

f(0);         // calls 2
f(nullptr);   // calls 1
```

Reference

- In modern languages definitions hide two orthogonal concepts:
  - Allocate memory for a variable
  - Bind a name with special scope to it
- In most languages this is inseparable
- In C++ we can separate the two steps

```
void f()
{
  int i;                  // allocate memory, bind i as name
  int &i_ref = i;         // binds a new name
  int *iptr = new int;    // allocate memory, no binded name
  int &k = *iptr;         // binds a new name to unnamed int area
  delete iptr;            // memory invalidated name k still lives
  k = 5;                  // compiles, later run-time error
}                         // k goes out of scope
```

Pointer vs reference

- Pointers have extreme element: nullptr
  - nullptr means: pointing to no valid memory
- References always should refer to valid memory
  - Use exception, if something fails

```
if ( Derived *dp = dynamic_cast<Derived*>(bp) )
{
    // use dp as Derived*
}

try
{
  Derived &dr = dynamic_cast<Derived&>(*bp); // may throw
  // use dr as Derived&
}
catch(bad_cast &e) { . . . }
```

Parameter passing

- Parameter passing in C++ follows initialization semantics
  - Value initialization copies the object
  - Reference initialization just set up an alias

```
void f1( int x, int y) { ... }
void f2( int &xr, int &yr) { ... }

int i = 5, j = 6;

f1( i, j);   int x = i;  // creates local x and copies i to x
             int y = j;  // creates local y and copies j to y

f2( i, j);   int &xr = i; // binds xr name to i outside
             int &yr = j; // binds yr name to j outside
```

Swap before move semantics

```
void swap( int &x, int &y)
{
  int tmp = x;
  x = y;
  y = tmp;
}

int main()
{
  int i = 5;
  int j = 6;

  swap( i, j);
  assert(i==6 && j==5);
}
```

Reference binding

- Non-const (left) reference binds only to left value
- Const reference binds to right values too

```
int i = 5, j = 6; double d = 7.0;

swap(i,j);      // ok
swap(i,7);      // error: could not bind reference to 7
int &ir1 = 7;  // error: could not bind reference to 7

swap(i,d);              // error: int(d) creates non-left value
int &ir2 = int(3.14); // error: int(3.14) creates non-left value

const int &ir3 = 7;          // ok, lifetime extension
const int &ir4 = int(3.14); // ok, lifetime extension
```

Returning with reference

- By default C++ functions return with copy
- Returning reference just binds the function result to an object

```
int f1()
{
  int i = 5;
  return i;   // ok, copies i before evaporating
}

int &f2()
{
  int i = 5;
  return i;   // oops, binds to evaporating i
}
```

Usage example

```
class date
{
public:
  date& setYear(int y)  { _year = y;  return *this; }
  date& setMonth(int m) { _month = m; return *this; }
  date& setDay(int d)   { _day = d;   return *this; }

  date& operator++() { ++_day; return *this; }
  date  operator++(int) // should return temporary
                { date curr(*this); ++_day; return curr; }
private:
  int _year;
  int _month;
  int _day;
};

void f()
{
  date d;
  ++d.setYear(2011).setMonth(11).setDay(11); // still left value
}
```

Usage example

```
template <typename T>
class matrix
{
public:
      T& operator()(int i, int j)       { return v[i*cols+j]; }
const T& operator()(int i, int j) const { return v[i*cols+j]; }
    matrix& operator+=( const matrix& other)
    {
        for (int i = 0; i < cols*rows; ++i)
            v[i] += other.v[i];
        return *this;
    }
private:
    // ...
    T* v;
};
template <typename T> matrix<T>  // returns value
operator+(const matrix<T>& left, const matrix<T>& right)
{
    matrix<T> result(left);  // copy constructor
    result += right;
    return result;
}
```

Left vs right value

- Assignment in earlier languages work the following way:
  <variable> = <expression>, like x = a+5;

- In C/C++ however it can be:
  <expression> = <expression>, like *++ptr = *++qtr;

- But not all expressions are valid, like a+5 = x;

  An **lvalue** is an expression that refers to a memory location and allows us to take the address of that memory location via the & operator. An **rvalue** is an expression that is not an lvalue

- A rigorous definition of lvalue and rvalue:
  https://accu.org/journals/overload/12/61/kilpelainen_227/

Value categories

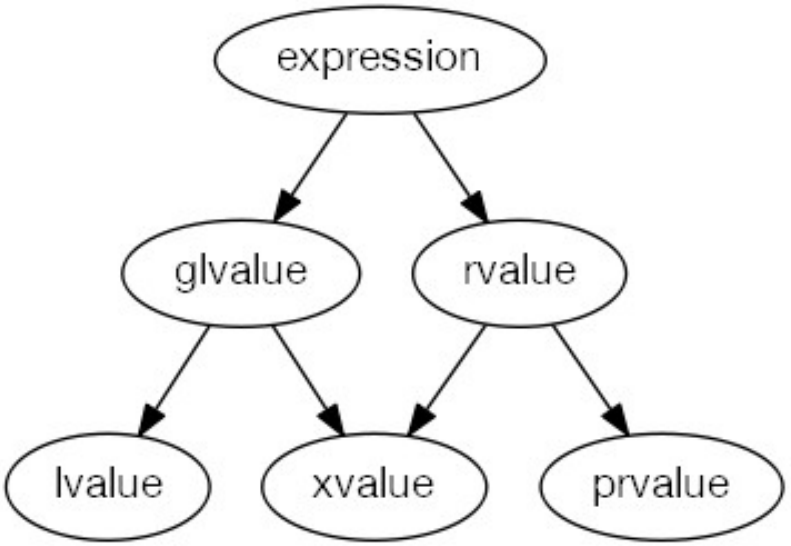
— An lvalue (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue. —end example]

— An xvalue (an “eXpiring” value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). An xvalue is the result of certain kinds of expressions involving rvalue references (8.3.2). [Example: The result of calling a function whose return type is an rvalue reference is an xvalue. —end example]

— A glvalue (“generalized” lvalue) is an lvalue or an xvalue.

— An rvalue (so called, historically, because rvalues could appear on the right-hand side of an assignment expressions) is an xvalue, a temporary object (12.2) or subobject thereof, or a value that is not associated with an object.

— A prvalue (“pure” rvalue) is an rvalue that is not an xvalue. [Example: The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as 12, 7.3e5, or true is also a prvalue. —end example]



Left value vs. right value

```
int  i = 42;
int &j = i;
int *p = &i;

 i = 99;
 j = 88;
*p = 77;

int *fp() { return &i; } // returns pointer to i: lvalue
int &fr() { return i; }  // returns reference to i: lvalue

*fp() = 66;  // i = 66
 fr() = 55;  // i = 55

// rvalues:

int f() { int k = i; return k; } // returns rvalue

  i = f();  // ok
  p = &f(); // bad: can't take address of rvalue
f() = i;    // bad: can't use rvalue on left-hand-side
```

Const correctness and pointers

```
      int i = 4;   // not constant, can be modified:
          i = 5;

const int ci = 6;   // const, must be initialized
          ci = 7;   // syntax error, cannot be modified

     int *ip;
          ip = &i;
         *ip = 5;   // ok

         if ( input )
           ip = &ci; // ??

         *ip = 7;   // can I do this?
```

Pointer to const

```
        int i = 4;   // not constant, can be modified:
            i = 5;

const int ci = 6;   // const, must be initialized
          ci = 7;   // syntax error, cannot be modified



const int *cip = &ci; // ok
          *cip = 7;   // syntax error
       int *ip = cip; // syntax error, C++ keeps const

           cip = ip;  // ok, constness gained
          *cip = 5;   // syntax error,
                      //    anywhere cip points to
```

Pointer constant

```
int i = 4;   // not constant, can be modified:
    i = 5;

const int ci = 6;   // const, must be initialized
    ci = 7;   // syntax error, cannot be modified
```

- `int * const ipc = &i; // ipc is const, must initialize`
  `*ipc = 5;  // OK, *ipc points is NOT a const`

```
int * const ipc2 = &ci; // syntax error,
                        // ipc is NOT a pointer to const

const int * const cccp = &ci; // const pointer to const
```