Basic C++

4

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

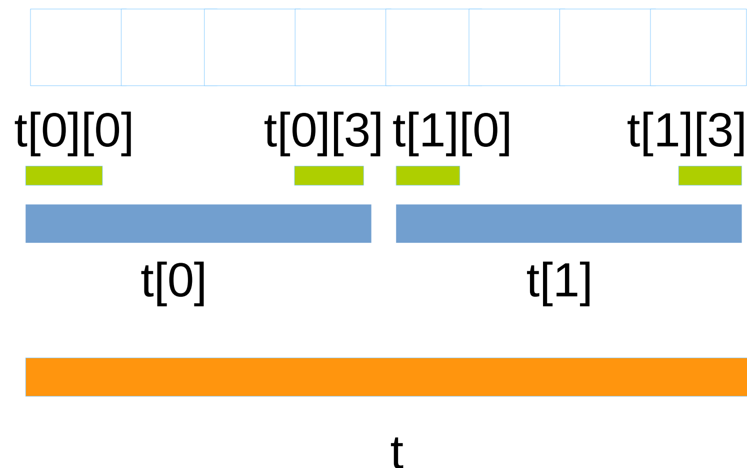
http://gsd.web.elte.hu

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
}
```

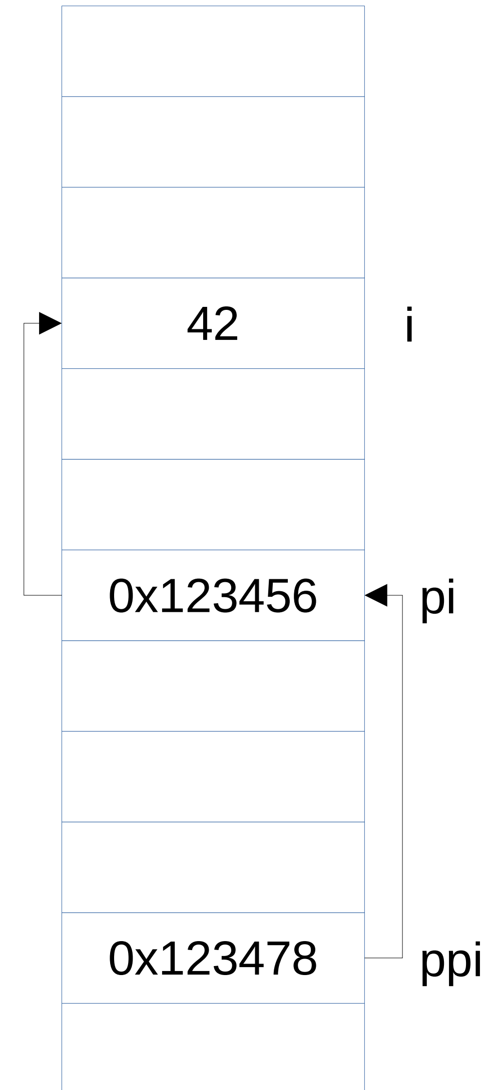
std::array

```
#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
}
```

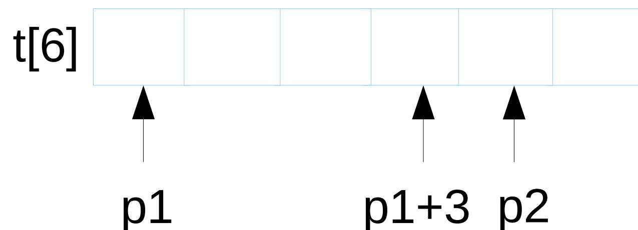
```
char *ptr = getchar("PATH");
if ( ptr )
{
   // *ptr can be used here
}
```

```
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

COMPILER EXPLORER Add... ▾ More ▾ Support diversity in C++ with #include <C++> ×

C++ source #1 ×   C++

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

int main()
{
  int *p = t;
  int x = t[1];
  int y = p[1];
}
```

x86-64 gcc 9.2 (Editor #1, Compiler #1) C++ ×

x86-64 gcc 9.2   Compiler options...

☐ 11010 ☐ ./a.out ☑ .LX0: ☐ lib.f: ☑ .text ☑ // ☐ \s+ ☑ Inte

```asm
t:
        .long   1
        .long   2
        .long   3
main:
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], OFFSET FLAT:t
        mov     eax, DWORD PTR t[rip+4]
        mov     DWORD PTR [rbp-12], eax
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax+4]
        mov     DWORD PTR [rbp-16], eax
        mov     eax, 0
        pop     rbp
        ret
```

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

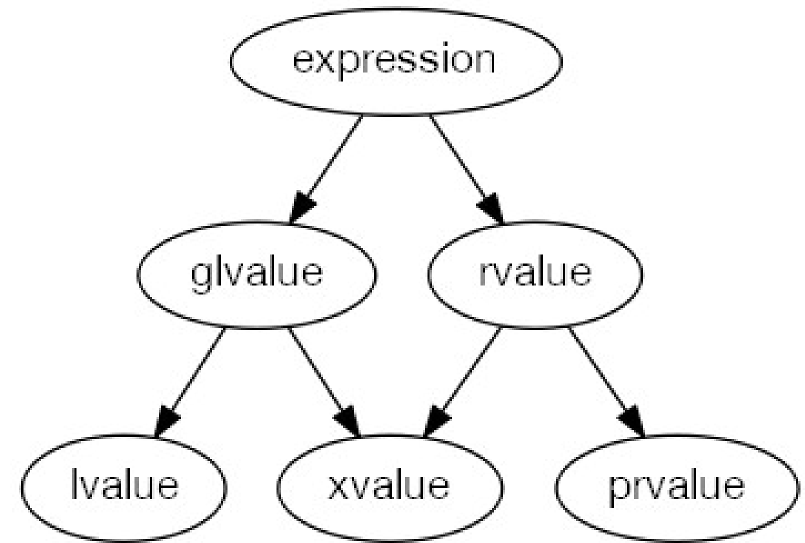
— An lvalue (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another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

Constants

- Constants in C++
- Const correctness
- Pointers to const
- Const member functions
- Const and mutable members
- STL const safety
- Constexpr

Const literals

```
char t1[] = {'H','e','l','l','o','\0'};
char t2[] = "Hello";
char t3[] = "Hello";

/* const */ char *s1 = "Hello"; // s1 points to 'H'
/* const */ char *s2 = "Hello"; // s2 points to the same place

// assignment to array elements ok:
*t1 = 'x'; *t2 = 'q'; *t3 = 'y';

// modifying string literal: can be segmentation error:
*s1 = 'w'; *s2 = 'z';
```

Preprocessor macro names

```
#define LOWER  -100
#define UPPER   400
#define STEP     40

int main()
{
  for( int fahr = LOWER; fahr <= UPPER; fahr += STEP )
  {
    std::cout << "fahr = " << std::setw(4) << fahr
          << ", cels = " << std::fixed << std::setw(7)
          << std::setprecision(2) << 5./9. * (fahr-32)
          << std::endl;
  }
  return 0;
}
```

Named constants

- ```cpp
  int f(int i) { return i; }
  int main()
  {
    const int c1 = 1;    // initialized compile time
    const int c2 = 2;    // initialized compile time
    const int c3 = f(3); // f() is not constexpr
    int t1[c1];
    int t2[c2];
    int t3[c3]; // VLA (Variadic Length Aray) is C99 not C++
    switch(i)
    {
    case c1: std::cout << "c1"; break;
    case c2: std::cout << "c2"; break;
    // case label c3 does not reduce to an integer constant
    case c3: std::cout << "c3"; break;
    }
    return 0;
  }
  ```
  ```

When named const needs memory?

```
int f(int i) { return i; }

int main()
{
  const int   c1 = 1;    // initialized compile time
  const int   c2 = 2;    // initialized compile time
  const int   c3 = f(3); // f() is not constexpr
  const int *ptr = &c2;
  return 0;
}
```

- Named pointers require storage when
  - Initialized at run-time (or)
  - Address is used

Be careful with optimization!

```
int main()
{
    const int ci = 10;
    int *ip = const_cast<int*>(&ci);
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
```

Be careful with optimization!

```
int main()
{
    const int ci = 10;
    int *ip = const_cast<int*>(&ci); // undefined behavior
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
10 11
```

Be careful with optimization!

```
int main()
{
    volatile const int ci = 10;
    int *ip = const_cast<int*>(&ci); // undefined behavior
    ++*ip;
    cout << ci << " " << *ip << endl;
    return 0;
}
```

```
$ ./a.out
11 11
```

Const correctness

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (*p = 0) ) ++p;
    return p - s;
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (*p = 0) ) ++p; // FATAL ERROR
    return p - s;
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(const char *s)
{
    const char *p = s;
    while ( ! (*p = 0) ) ++p; // Compile-time error!
    return p - s;
}


int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Const correctness

```
#include <iostream>

int my_strlen(const char *s)
{
    const char *p = s;
    while ( ! (*p == 0) ) ++p; // FIX
    return p - s;
}

// this program is fine
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

Side note on defensive programming

```
#include <iostream>

int my_strlen(char *s)
{
    char *p = s;
    while ( ! (0 = *p) ) ++p; // compile-time error!
    return p - s;             // “Yoda condition”
}

// This program likely cause run-time error
int main()
{
    char t[] = "Hello";
    std::cout << my_strlen(t) << std::endl;
    return 0;
}
```

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
```

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
```
```

const – pointer cheat sheet

- **const** keyword left to * means pointer to **const**
  - Can point to **const** variable
  - Can be changed
  - Pointed element is handled as constant

```
const int ci = 10;
const int * cip = &ci;
int const * pic = &ci;
```

- **const** keyword right to * means pointer is constant
  - Must be initialized
  - Can not be changed
  - Can modify pointed element

```
int i;
int * const ipc = &i;
```

Constexpr objects in C++11

- Const objects having value known at translation time.
- translation time = compilation time + linking time
- They may have placed to ROM
- Immediately constructed or assigned
- Must contain only literal values, constexpr variables and functions
- The constructor used must be constexpr constructor

Constexpr functions in C++11/14

- Can produce constexpr values when called with compile-time constants.
- Otherwise can run with non-constexpr parameters
- Must not be virtual
- Return type must be literal type
- Parameters must be literal type
- Since C++14 they can be more than a return statement
  - if / else / switch
  - for / ranged-for / while / do-while

Constexpr in C++11

```
enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2)
{
  return Flags(int(f1)|int(f2));
}

void f(Flags x)
{
  switch (x)
  {
  case bad:         /* ... */ break;
  case eof:         /* ... */ break;
  case bad|eof:     /* ... */ break;
  default:          /* ... */ break;
  }
}
```

Constexpr in C++11

```
include <cstdio>

size_t constexpr length(const char* str)
{
    return *str ? 1 + length(str + 1) : 0;
}

void f()
{
    printf("%d %d", length("abcd"), length("abcdefgh"));
}
```

- Pattern matching + recursion == Turing complete

Constexpr in C++14

```
#include <iostream>

constexpr int strlen(const char *s)
{
    const char *p = s;
    while ( '\0' != *p )  ++p;
    return p-s;
}
int main()
{
    std::cout << strlen("Hello") << std::endl;
    return 0;
}

constexpr int pow( int base, int exp) noexcept
{
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
}
```

Constexpr in C++20

- Union (also needed for constexpr string)
- Try and catch (throw not allowed, so catch works only runtime)
- dynamic_cast and typeid (since we have virtual functions)
- Constexpr allocation
  - Transient: deallocated before evaluation completes
  - Non-transient: not (yet)
- Virtual calls in constexpr, constexpr virtual funtions/overrides
- Library: constexpr vector and string

Constexpr in C++20

```
// C++20

constexpr auto naiveSum(unsigned int n)
{
    auto p = new int[n];  // transient allocation C++20

    // iota is constexpr in C++20
    std::iota(p, p+n, 1);

    // accumulate is constexpr in C++20
    auto tmp = std::accumulate(p, p+n, 0);

    delete [] p;    // compiler detects delete/delete[] issues
    return tmp;
}
```

Functions

- Fundamental building blocks
  - Should be declared as local or global
  - Should be defined as global (no nested functions)
- Function associates
  - Function name
  - Block: sequence of statements
  - Return type (or void)
  - Parameters (can be defaulted)

Function call

- Functions should be declared before first use
- Should be declared as local or global
  - Should be defined as global (no nested functions)
- Function associates
  - Function name
  - Block: sequence of statements
  - Return type (or void)
  - Parameters (can be defaulted)

Functions

```
bool isodd( int n); // functions should be declared before first use

int f()
{
    for ( auto arg : { 1, 3, 42, -5, 8} )    // arg is int
    {
        std::cout << isodd(arg) << ' '; // function call: arg copied to param
    }
}

bool isodd( int n)  // arg is copied to n for every call
{
    return n % 2;
}
```

Functions

```
bool isodd( int n); // formal parameters can be ignored

int f()
{
    for ( auto arg : { 1, 3, 42, -5, 8} )    // arg is int
    {
        std::cout << isodd(arg) << ' '; // function call: arg copied to param
    }
}

bool isodd( int n)  // arg is copied to n for every call
{
    return n % 2;
}
```

Functions

```
bool isodd( int ); // functions should be declared before first use

int f()
{
    for ( auto arg : { 1.5, 3.14, 42.0, -5.5, 8.8} )  // arg is double
    {
        std::cout << isodd(arg) << ' '; // function call: arg copied to param
    }
}

bool isodd( int n)  // arg is converted to int then copied to n
{
    return n % 2;
}
```

Functions

```
bool isodd( int ); // functions should be declared before first use

int f()
{
    for ( auto arg : { "1", "3", "42", "-5", "8"} )  // arg is const char *
    {
        std::cout << isodd(arg) << ' '; // compile error
    }                                   // arg does not convertible to int
}

bool isodd( int n)
{
    return n % 2;
}
```

Function declaration

```
int f( int i, double d, std::string s);  // function declaration
int f( int  , double  , std::string  );  // same as above

int g();        // function with no parameter
int g(void);    // C style declaration, same as above

void h( int, double, std::string);  // function with no return type

int printf( const char *format, ...)  // one or more parameters
int fprintf( FILE *fp, const char *format ...)  // two or more parameters
int fdots(...)  // zero or more parameters, we do know nothing about the pars

template <typename ...Args>    // variadic function template
void tf( Args ...pars);

date_t make_date( int year, int month = 1, int day = 1); // default arguments
```

Default arguments

- Default arguments belong to the caller environment
- Should be continuous from right-to-left
- Can be different in independent scopes

```
void f()
{
  date_t make_date( int year, int month = 1, int day = 1);  // default args

  date_t d1 = make_date( 2023, 8, 14);
  date_t d1 = make_date( 2023, 8);  // make_date( 2023, 8, 1)
  date_t d1 = make_date( 2023);     // make_date( 2023, 1, 1)
}
void g()
{
  date_t make_date( int year, int month = 12, int day = 25);
  date_t xmas = make_date( 2023);   // make_date( 2023, 12, 25)
}
```

Function definition

```
double fahr2cels( double f)
{
    return 5./9.*(f-32);
}

auto cels2fahr( double c)  // return type deduction
{
    return c*9./5.+32;
}

void f( double par1, int )    //  unused parameters may not named
{
    return fahr2cels(par1);
}

int fact( int f)  // recursive function
{
    if ( 1 == n )
        return 1;
    else
        return n*fact(n-1);    // tail recursion, can be optimized
}
```

Parameter passing by value

```
#include <iostream>

void increment(int par)    // creates a copy of actual argument of i in main
{
  ++par;   // increments local copy
  std::cout << "i in increment() " << par << std::endl;
}

int main()
{
  int i = 0;
  increment(i);  // copies 0 to par
  increment(i);  // copies 0 to par
  std::cout << "i in main() = " << i << std::endl;
  return 0;
}

$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 1
i in main() = 0
```

Parameter passing by “address”

```
#include <iostream>

void increment(int *par)   // par points to the memory area of i in main
{
  ++*par;  // increments i in main
  std::cout << "i in increment() " << *par << std::endl;
}

int main()
{
  int i = 0;
  increment(&i);  // copies the address of i to par
  increment(&i);  // copies the address of i to par
  std::cout << "i in main() = " << i << std::endl;
  return 0;
}

$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 2
i in main() = 2
```

Parameter passing by reference

```
#include <iostream>

void increment(int &par)   // par refers to the memory area of i in main
{
  ++par;   // increments i in main
  std::cout << "i in increment() " << par << std::endl;
}

int main()
{
  int i = 0;
  increment(i);  // initialize par (reference) to the memory area of i
  increment(i);  // initialize par (reference) to the memory area of i
  std::cout << "i in main() = " << i << std::endl;
  return 0;
}

$ g++ -ansi -pedantic -Wall -W par.cpp
$ ./a.out
i in increment() = 1
i in increment() = 2
i in main() = 2
```

Function overloading

- Overload resolution happens when function name is not unique
- Overload resolution process
  - Building candidate function set
  - Reduction to viable functions
  - Find the best viable function (if exists)

Overload resolution 1

- Candidate functions
  - Finding the callable entities which match to the call expression
- Viable functions
  - The same number of parameters as in the call expression
  - Less parameters, but there is ellipsis parameter (...)
  - Less parameters, but the rest has/have default arguments
  - Constraints are satisfied (C++20)
  - Every argument is convertible to the corresponding parameter

Overload resolution 2

- Best viable function
  - Ranking the conversion from actual argument to formal one
  - If there is a parameter where the conversion is better and the all other parameters are not worse
- Ranking of conversion (best to worse)
  - No or trivial conversions
  - Arithmetic promotions
  - Static cast
  - User defined conversions

Overload resolution

```
void f(const int *, short);
void f(int *, int);

int main()
{
  int   i = 0;
  short s = 0;
  long  l = 10L;

  f( &i, l);   // (1) &i->int* > &i->const int* (2) l->int = l->short
               // calls f(int *, int)

  f( &i, 'c'); // (1) &i->int* > &i->const int* (2) 'c'->int > 'c'->short
               // calls f(int *, int)

  f( &i, s);   // (1) &i->int* > &i->const int* (2) s->int < s->short
               // compiler error
}
```

How to define operators?

- a + b, a – b, a == b, ...

  a.**operator**+(b)

  operator+(a,b)

- a = b, a[b], a(b1,b2,...), a-> only member

  a.**operator**= (b)

  a.**operator**[ ] (b)

  a.**operator**( ) (b1, b2, …)

  a.**operator**->()

Operators

```
#include <iostream>
#include <string>

int main()
{
    int i = 42;
    std::cout << i << '\n';  // std::cout.operator<<(i).operator<<('\n');
}
```

Operators

```
#include <iostream>
#include <string>

int main()
{
    int i = 42;
    std::cout << i << '\n';  // std::cout.operator<<(i).operator<<('\n');
}
```

```
int main()
{
    std::string h = "Hello ";
    std::string w = "world\n";
    std::cout << h << w;     // std::operator<<(operator<<(cout,w),s);
}
```

Operators

```
#include <iostream>

struct complex_t
{
    double re;
    double im;
};

std::ostream& operator<< ( std::ostream& os, const complex_t &c)
{
    os << re << '+' << im << 'i';
    return os;    // important for chaining output operations
}

int main()
{
    complex_t c{1, 3.14};
    std::cout << c << '\n'; // std::operator<<(operator<<(cout,c),'\n');
}
```

Operators

```
#include <iostream>
#include <fstream>

int main()
{
    complex_t c{1, 3.14};
    std::ofstream outfile{"out.txt};

    if ( outfile )
    {
        outfile << c << '\n';
    }
}
```

Operators

```
#ifndef COMPLEX_T
#define COMPLEX_T

#include <iostream>

struct complex_t
{
    double re;
    double im;
    bool operator==( const complex_t& c) const  // similarly != < etc.
    {
        return re == c.re && im == c.im;
    }
};

#endif // COMPLEX_T

int main()
{
    complex_t c1{1, 3.14}, c2{3.14,2};
    std::cout << (c1 == c2) << '\n';  // 0
}
```

Operators

```
#ifndef COMPLEX_T
#define COMPLEX_T

#include <iostream>

struct complex_t
{
    double re;
    double im;

};
bool operator==( complex_t c1, complex_t c2)  // similarly != < etc.
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_T

int main()
{
    complex_t c1{1, 3.14}, c2{3.14,2};
    std::cout << (c1 == c2) << '\n';  // 0
}
```

Operators

```cpp
#ifndef COMPLEX_T
#define COMPLEX_T

#include <iostream>

struct complex_t
{
    double re;
    double im;

};
bool operator==( complex_t c1, complex_t c2)  // similarly != < etc.
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_T

int main()
{
    complex_t c1{1, 3.14}, c2{3.14,2};
    std::cout << std::boolalpha << (c1 == c2) << '\n';  // false
}
```

Inline functions

```
#ifndef COMPLEX_T
#define COMPLEX_T

#include <iostream>

struct complex_t
{
    double re;
    double im;

};
inline bool operator==( complex_t c1, complex_t c2)  // similarly != < etc.
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_T

int main()
{
    complex_t c1{1, 3.14}, c2{3.14,2};
    std::cout << std::boolalpha << (c1 == c2) << '\n';  // false
}
```

Global declarations in C++

```
#include <iostream>
void f();
            int g0 = 10;
     inline int g1 = 11;
static inline int g2 = 12;
extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

```
#include <iostream>
            int g0 = 20;
     inline int g1 = 21;
static inline int g2 = 22;
extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Global declarations in C++

```
#include <iostream>
void f();
                int g0 = 10;
           inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

/usr/bin/ld: inline2.o:(.data+0x0): multiple definition of `g0'; inline1.o:(.data+0x0): first defined here

```
#include <iostream>
                int g0 = 20;
           inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Global declarations in C++

```
#include <iostream>
void f();
               int g0 = 10;
          inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

/usr/bin/ld: inline2.o:(.data+0x0): multiple definition of `g0'; inline1.o:(.data+0x0): first defined here

```
#include <iostream>
         extern int g0 = 20;
          inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Global declarations in C++

```
#include <iostream>
void f();
                int g0 = 10;
           inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

```
#include <iostream>
         extern int g0;
         inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

Global declarations in C++

```
#include <iostream>
void f();
               int g0 = 10;
           inline int g1 = 11;
    static inline int g2 = 12;
    extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

```
#include <iostream>
         extern int g0;
         inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

&g0 = 10, 0x40406c
&g1 = 21, 0x404060
&g2 = 12, 0x404070
&g3 = 23, 0x404068
&g0 = 10, 0x40406c
&g1 = 21, 0x404060
&g2 = 22, 0x404064
&g3 = 23, 0x404068

Global declarations in C++

```cpp
#include <iostream>
void f();
                int g0 = 10;
           inline int g1 = 11;
  static inline int g2 = 12;
  extern inline int g3 = 13;

int main()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
  f();
  return 0;
}
```

```cpp
#include <iostream>
           extern int g0;
           inline int g1 = 21;
  static inline int g2 = 22;
  extern inline int g3 = 23;

void f()
{
  std::cout << "&g1 = " << g1 << ", " << &g1 << '\n';
  std::cout << "&g2 = " << g2 << ", " << &g2 << '\n';
  std::cout << "&g3 = " << g3 << ", " << &g3 << '\n';
}
```

&g0 = 10, 0x40406c
&g1 = 11, 0x404060
&g2 = 12, 0x404070
&g3 = 13, 0x404068
&g0 = 10, 0x40406c
&g1 = 11, 0x404060
&g2 = 22, 0x404064
&g3 = 13, 0x404068

Linker symbols

```
$ readelf inline1.o

Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS inline1.cpp
     2: 0000000000000040    11 FUNC    LOCAL  DEFAULT    4 _GLOBAL__sub_I_inline1.cp
     3: 0000000000000004     4 OBJECT  LOCAL  DEFAULT    7 _ZL2g2
     4: 0000000000000000     1 OBJECT  LOCAL  DEFAULT    6 _ZStL8__ioinit
     5: 0000000000000000    59 FUNC    LOCAL  DEFAULT    4 __cxx_global_var_init
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    2
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _Z1fv
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSolsEPKv
    13: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSolsEi
    14: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSt8ios_base4InitC1Ev
    15: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZNSt8ios_base4InitD1Ev
    16: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZSt4cout
    17: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZStlsISt11char_traitsIcE
    18: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND _ZStlsISt11char_traitsIcE
    19: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND __cxa_atexit
    20: 0000000000000000     0 NOTYPE  GLOBAL HIDDEN   UND __dso_handle
    21: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    7 g0
    22: 0000000000000000     4 OBJECT  WEAK   DEFAULT   10 g1
    23: 0000000000000000     4 OBJECT  WEAK   DEFAULT   12 g3
    24: 0000000000000000   316 FUNC    GLOBAL DEFAULT    2 main
```