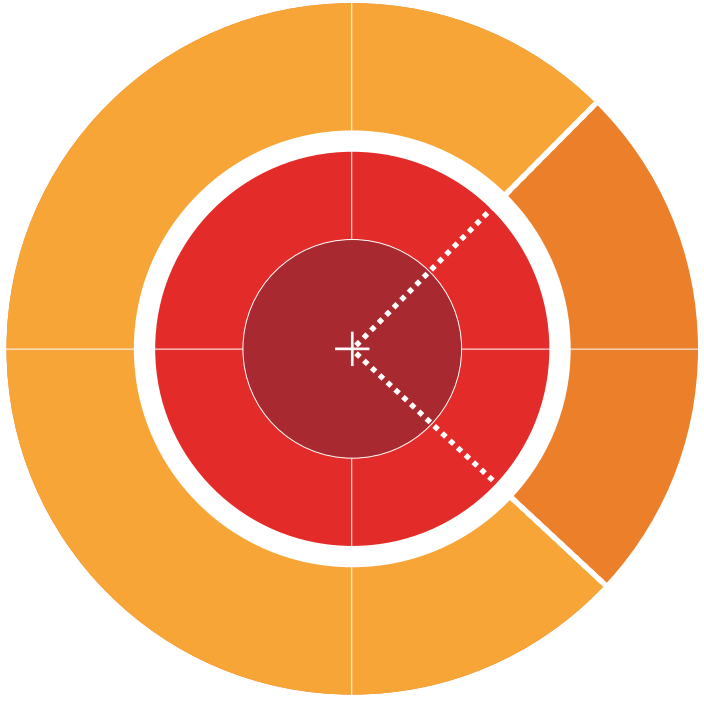
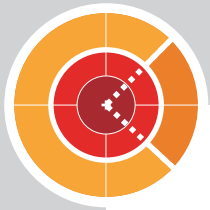




C++ [si: pl s pl s] - C s objekty

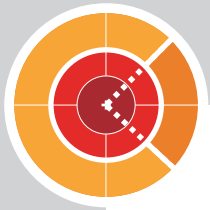
- rozšíření jazyka C, navržené **Bjarnem Stroustrupem** v Bell Labs v průběhu 80. let 20. století (C je vlastně podmnožinou C++)
- cílem rozšíření bylo zahrnout do jazyka C vlastnosti a prvky, které se v té době začínaly stávat populárními (zejména objekty, které se objevily ve Smalltalku a Simule-67)
- nejdříve byly přidány **třídy** (**objekty**), následně pak také virtuální funkce, přetěžování operátorů, vícenásobná dědičnost, šablony a zpracování výjimek
- standardizace započala zachycením stavu jazyka v roce 1998, poslední standard je z roku 2003

- výsledek je značně nepřehledný a **komplikovaný**, což se stalo hlavním důvodem k návrhu jazyků Java a C#.

Jazyk C++ je:

- stále nízkoúrovňový, umožňuje vytvářet efektivní a rychlý kód (rychlejší než C)
- velmi rozšířený, široce podporovaný
- velmi expresivní, rozšiřitelný, bezpečnější než C (type-checking)

- velmi nepřehledný, značně komplikovaný
- méně přenositelný než ANSI C
- různé "dialekty" podle překladačů



Překlad zdrojových kódů v C++

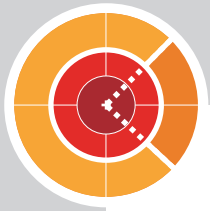
- většinou stejným překladačem - sám si zjistí, zda jde o C nebo C++ (v lepším případě podle syntaxe, v horším podle přípony ZK)
- v některých prostředích (Dev-C++) je třeba při zakládání projektu zaškrtnout příslušné políčko, zda se jedná o C či C++
- v řídkých případech se pak C++ překládá jiným překladačem (např. Open Watcom má **`wpp386`**, ale pokud voláme překladový systém **`cl`** / **`wcl386`**, zavolá si ho sám aniž by to musel programátor řešit)

```
E:\Work>wpp386 hello.cpp
```

```
E:\Work>wcl386 hello.cpp
```

```
E:\Work>wcc386 hello.cpp
```

tohle fungovat nebude

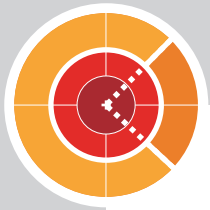


Změny oproti ANSI C

Nový druh komentáře

```
/* tohle je klasický (jediný správný)
   komentář v ANSI C */

// tohle je nový komentář v C++
// nemá ukončovací značku, končí vždy
// koncem řádky
```

- komentářem je vše od výskytu lexikálního atomu `//` do konce řádky, pokud má být komentář víceřádkový, je třeba na každé nové řádce vždy uvést `//`
- pozitivní změna, `//` nekoliduje s žádným operátorem ani jejich kombinací, nevýhodou je skutečnost, že neexistuje ukončovací znak, takže nelze komentovat "vnitřek" kódu



Deklarace konstantních proměnných

```
#define PI 3.14159265358979323846
```

- ANSI C umožňuje definovat konstanty prakticky pouze mechanismem preprocesoru

```
const double PI = 3.14159265358979323846;
```

modifikátor const oznamuje překladači, že se proměnná nesmí (a nebude) měnit

- konstanta může být inicializovaná při deklaraci, ale dále se nesmí měnit => překladač může generovat výkonnější kód
- každá konstanta má současně pevně daný datový typ, takže překladač může lépe provádět typovou kontrolu

```
int i = 10;
const int *pi = &i;

*pi = 15;
```

nelze! ukazatel na `const int`

Využití ukazatelů na konstanty

```
int i = 10;
const int *pi = &i;

*pi = 15;
```

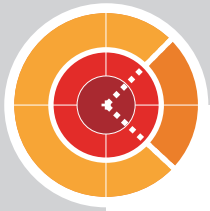
hodnotu `i` nelze dereferencí ukazatele změnit

- deklaruje se vlastně read-only ukazatel
- konstantní ukazatele se často používají jako návratové typy funkcí, které vrací ukazatel na nějakou oblast paměti, kterou lze číst, ale ne modifikovat (např. proměnné instancí tříd deklarované jako *private*)

- C++ má stále mnoho vlastností C => "konstantnosti" se lze zbavit vhodným přetypováním

```
*((int *) pi) = 15;
```

tzv. casting away the const-ness



Deklarace konstantních ukazatelů

```
int i = 10;
int j = 11;
int * const ptr = &i;

ptr = &j;

*ptr = 15;
```

ukazatel **ptr** je konstantní, tj. nesmí ukazovat na nic jiného, než na co je zinicializovaný

nelze!

bez problému - objekt, na který ukazatel ukazuje **není konstantní**

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

konstantní ukazatel na konstatní objekt

- **const** zlepšuje typovou kontrolu při překladu
- narozdíl od konstant zavedených pomocí **#define** se tyto **konstanty ukládají do informací pro debugger**

Inline funkce

- funkce deklarované s modifikátorem **`inline`** se přímo vkládají na místo výskytu => urychlení, neprovádí se obvyklé postupy při volání podprogramu (uložení registrů, operace se zásobníkem, atp.); **pro krátké jednoduché funkce**

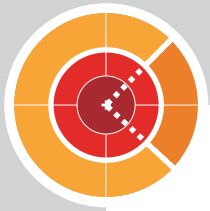
```
#define max (x, y) ((x)>(y)?(x):(y))

inline int max(int x, int y) {
  return (x > y ? x : y);
}
```

ANSI C

- vhodná náhrada málo bezpečných maker preprocesoru

```
max(f(x), z++);

((f(x)) > (z++) ? (f(x) : (z++));
```

- při použití **`inline`** tento případ nehrozí



Předávání parametrů odkazem

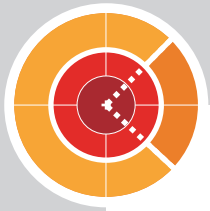
- C++ umožňuje předávat funkcím parametry odkazem (jako Pascal, Ada, apod.)

```
void swap(int &a, int &b) {
   int tmp = a;
   a = b;
   b = tmp;
}
...
int n = 10;
int &r = n;
r = -10;
...
int x = 1, y = 2;
swap (x, y);
```

deklarace odkazu

r je tzv. **alias - musí mít vždy inicializační část**, vytváří spojení mezi odkazem a objektem, na který odkazuje...

přiřazení změní hodnotu **n** na 10

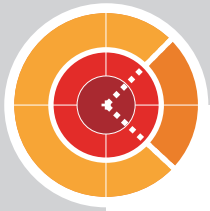


Využití odkazu jako návratového typu funkce

```
int &GetItem(int *array, int index) {
  return array[index];
}

...

int A[100];

for (int i = 0; i < 100; i++)
  GetItem(A, i) = 25;
```

přiřazení je v pořádku, je to L-value, protože funkce vrátila adresu

- bude-li funkce deklarována jako **`inline`**, eliminuje se režie spojená s voláním podprogramu => inline funkce vracející odkaz jsou velmi užitečné ke zvýšení výkonu



Deklarace proměnných jako příkaz

- v C++ je deklarace proměnné příkaz, nemá zvláštní postavení jako v ANSI C, a proto může být použita **kdekoliv**

```
int IsMember(const int key) {
  int found = 0;
  if (NotEmpty()) {
    List* ptr = head;
    while (ptr && !found) {
      int item = ptr->data;
      ptr = ptr->next;
      if (item == key) found = 1;
    }
  }
  return found;
}
```

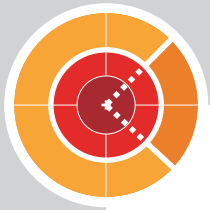
- uvádění deklarací blíž výskytu proměnné zvyšuje čitelnost ZK, nebezpečím je možná nesprávná interpretace

Vylepšení typového systému pomocí tříd

```
double a[3] = { 11, 12, 13 };
double b[3] = { 21, 22, 23 };

Vector v1 = a;
Vector v2 = b;

Vector v3 = v1 + v2;
double norm = (double) v3;
```

je-li správně nadefinována třída **`Vector`**, bude se k ní překladač chovat jako k datovému typu...

- předpokladem je definice třídy **`Vector`** se všemi použitými operátory (viz dále) a konverzemi - pak s ní bude překladač zacházet jako s novým datovým typem



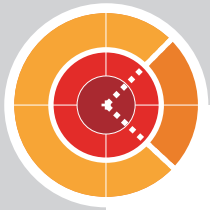
Datový typ bool

- C++ má nový vestavěný datový typ: **`bool`**, který může nabývat hodnoty **`true`** nebo **`false`** (logický typ)

```
bool flag = true;
…
flag = false;
…
if (flag) { … }
```

při testování v podmínce if se samozřejmě zkoumá shoda s **`true`**

- v podmínkách přirozenější (čitelnější a jednodušší) než v ANSI C, kde hrál celočíselný typ i roli logického



Přetěžování funkcí

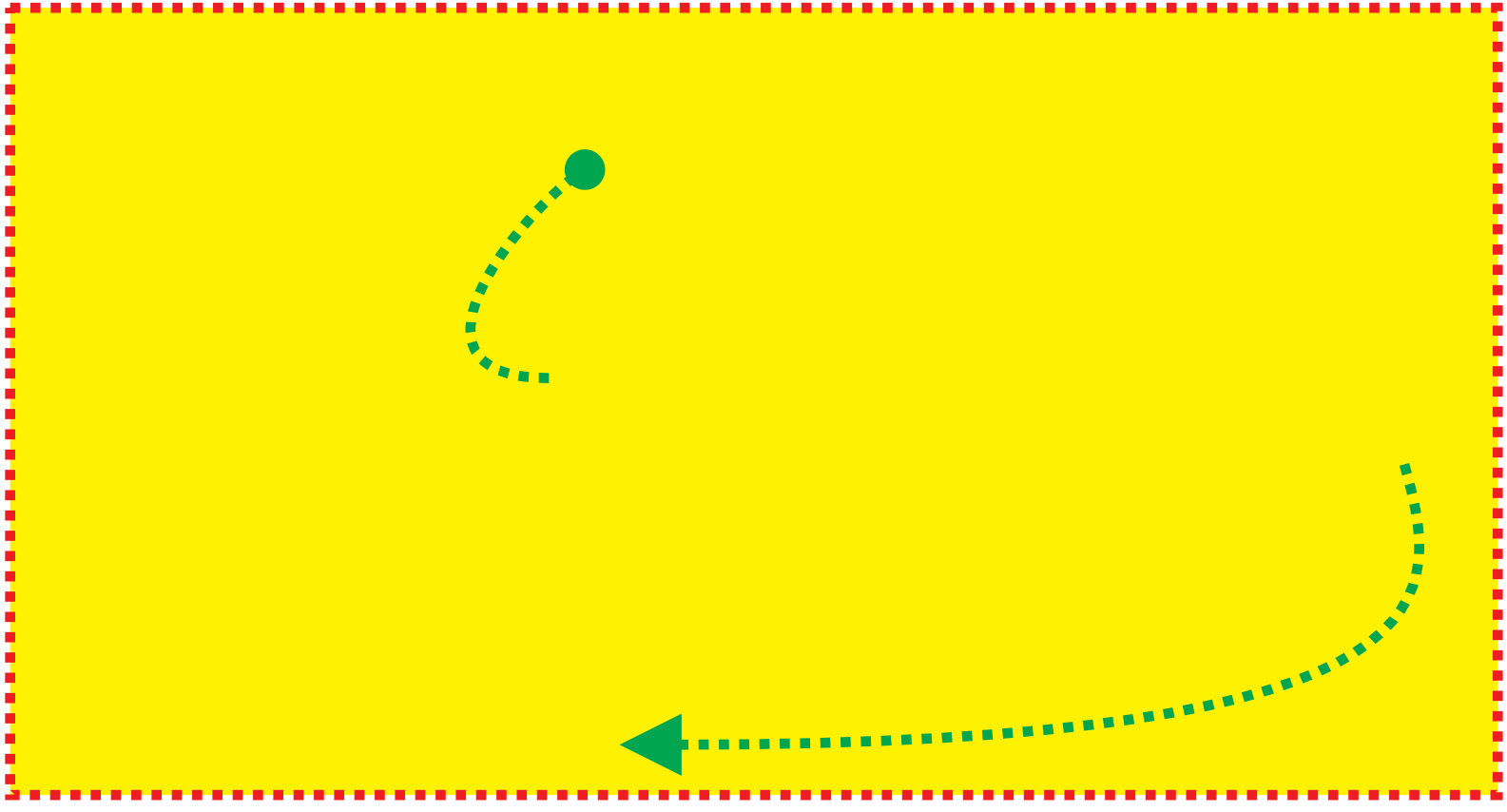
- C++ dovoluje **přetěžovat** funkce, tj. definovat několik funkcí stejného jména (se stejnou logikou) pro různé datové typy

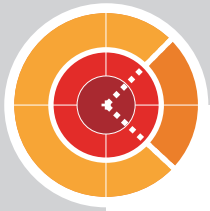
```
int find(const int* data, const int key) {

}

int find(const float* data, const float key) {

}

int pos, a[] = {1, 2, 3, 4, 5, 6, 7, 8};
float b[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};

pos = find(a, 5);
pos = find(b, 5.0);
```

překladač zavolá tu funkci, která vyhovuje předávaným p-metrům

Přetěžování operátorů

- C++ dovoluje **přetěžovat** také operátory, tj. definovat význam/výpočet operace dané operátorem pro daný datový typ





Vytváření instancí alokací paměti

- C++ má dva nové operátory, **new** a **delete**, které slouží k vytváření, resp. odstraňování instancí z paměti (heap)

```
int *p = NULL;
p = malloc(sizeof(int));
*p = 10;
```

VS

```
int *p = NULL;
p = new int;
*p = 10;
```

deklarace dynamického pole (**ne** statického) a jeho alokace v inicializační části

místo **malloc()**

```
int *array = new int[10];

for (int i = 0;i < 10; i++)
  array[i] = i;

delete [] array;
```

odstranění dynamického objektu z paměti, **[]** říká překladači, aby delete aplikoval na všechny prvky

Používání jmenných prostorů - namespace

- C++ zavádí zcela nový mechanismus, tzv. *jmenné prostory* (namespaces)

```
namespace mynamespace {
   int i;
}
```

takto vznikne jmenný prostor pojmenovaný **mynamespace** - v něm je deklarovaná proměnná **i**

- mimo definovaný jmenný prostor je proměnná přístupná pouze úplnou kvalifikací

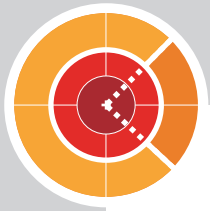
```
mynamespace::i = 5;
```

operátor přístupu k členské proměnné

```
using namespace mynamespace;

i = 5;
```

nejednoznačnost

```
using namespace mynamespace;
int i;
i = 5;
```



Odkazy na jména ze jmenných prostorů

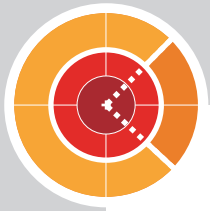
```
namespace nsp1 {
   namespace nsp2 {
      double pi;
   }

   using nsp2::pi;
   double x = pi;
}

double i = nsp1:nsp2:pi;
```

proměnná **pi** se teď bere z namespace **nsp2**

zde už **using** nepůsobí, tj. je třeba uvádět **plně kvalifikované jméno**

- jmenné prostory jsou otevřené, lze je **redeklarovat**

```
namespace mysp { int i; double pi; }

namespace mysp { int x, y, z; }
```



Speciální jmenné prostory

```
namespace {
   int x;
}
```

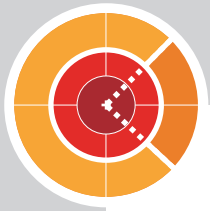
nepojmenovaný jmenný prostor deklaruje jména, která jsou v rámci daného zdrojového souboru nebo jiného **`namespace` neveřejná**

```
namespace $$$ {
   int x;
}

using namespace $$$;
```

`$$$` říká překladači, aby vygeneroval **unikátní ne-konfliktní identifikátor**

```
namespace ns_with_very_long_name {
   int i; double pi;
}

namespace ns = ns_with_very_long_name;
```

jmenné prostory lze **přiřazovat**



Jmenný prostor std

```
using namespace std;
```

- tradiční “úvodní” příkaz v C++
- říká překladači, že jakýkoliv identifikátor, který není definovaný ve ZK má hledat ve **standardním jmenném prostoru** (součást knihovny)

Připojování knihoven

- stejně jako v ANSI C, ale bez přípony headeru

```
#include <stdio>
#include <cstdio>
```

- standardní headery C++ jsou doplněny o konverzní knihovny (vycházejí z ANSI C, jsou upravené pro C++)

Vstupní/výstupní proudy - knihovna iostream

```
#include <iostream>
using namespace std;

int main (int argc, char *argv[]) {
  char quit = '\0';
  while (quit != 'q') {
    cout << "Hello!" << endl;
    cout << "Press q to quit " << endl;
    cin >> quit;
  }

  return 0;
}
```

znak(y) nové řádky

operátor přesměrování proudu

vstupní proud svázaný se **stdin**
výstupní proud svázaný se **stdout**

- do proudu lze "poslat" i jiný datový typ než řetězec znaků, ale musí mít definovanou metodu pro zápis do proudu